

STRUCTUREZ VOS DONNÉES AVEC

XML

Ludovic Roland



DANS LA MÊME COLLECTION D'OPENCLASSROOMS



Apprenez à créer votre site web avec HTML5 et CSS3

Mathieu Nebra

ISBN : 978-2-9535278-8-9



Concevez votre site web avec PHP et MySQL • 2e édition

Mathieu Nebra

ISBN : 979-10-90085-4-11



Apprenez à programmer en C • 2e édition

Mathieu Nebra

ISBN : 979-10-90085-00-8



Programmez avec le langage C++

M. Nebra et M. Schaller

ISBN : 979-2-9535278-5-8



Apprenez à programmer en Java

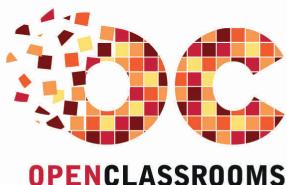
Cyrille Herby

ISBN : 979-10-90085-07-7

STRUCTUREZ VOS DONNÉES AVEC

XML

Ludovic Roland





Sauf mention contraire, le contenu de cet ouvrage est publié sous la licence :
Creative Commons BY-NC-SA 2.0

La copie de cet ouvrage est autorisée sous réserve du respect des conditions de la licence
Texte complet de la licence disponible sur : <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/>

Mentions légales :

Conception couverture : Fan Jiyong
Illustrations chapitres : Fan Jiyong

OpenClassrooms 2014 - ISBN : 979-10-90085-57-2

Avant-propos

À près une orientation post-bac décevante, des amis m'ont fait découvrir l'informatique en 2008 et m'ont conseillé le Site du Zéro, devenu aujourd'hui OpenClassrooms. Comme beaucoup de « Zéros » de l'époque, je me suis tourné vers le cours de C écrit par Mathieu NEBRA, lequel, avec le recul, était étrangement accessible en dépit de la difficulté de certaines notions abordées. Tout débutant en C se souvient forcément de sa rencontre avec les pointeurs !

Séduit par l'informatique, j'ai décidé de me réorienter dans ce secteur en intégrant le DUT Informatique de l'université d'Orsay, ce qui m'a permis d'appréhender de nombreux domaines et technologies, allant de la programmation au réseau, en passant par les bases de données. Pendant toute la durée de mon DUT, OpenClassrooms et sa communauté m'ont été d'une grande aide. Pour tenter de les remercier, j'ai posté mes premiers commentaires sur le forum du site, puis j'ai débuté la rédaction de mon premier tutoriel sur certains modules du langage Perl.

Après l'obtention de mon DUT, j'ai intégré une école d'ingénieur (l'EFREI) où j'ai suivi la filière Système d'Information, option Architecture des Systèmes d'Information. J'y ai appris de nouveaux aspects de l'informatique comme le JEE ou encore le Cloud Computing. Parallèlement à mes études, j'ai découvert, avec l'émergence des mobiles, la programmation Android et Windows Phone et ainsi, les problématiques d'échanges de données entre les serveurs et ces nouveaux clients. Le XML étant une solution pour structurer les données afin d'effectuer ces fameux échanges, j'ai commencé à m'intéresser à cette technologie en même temps qu'au développement de mes premières applications. Fraîchement diplômé de l'EFREI en 2013, j'ai décidé de faire du développement mobile mon métier.

OpenClassrooms ne proposant pas de cours complet sur le XML, en juin 2012, j'ai décidé d'écrire mon second tutoriel pour le site. Ce tutoriel a été mis en avant sur le site et est aujourd'hui adapté en livre. Je suis heureux de pouvoir vous proposer ce cours et j'espère de tout cœur qu'il répondra à vos attentes !

Comment lire ce livre ?

Suivez l'ordre des chapitres

Lisez ce livre comme on lit un roman. Il a été conçu pour cela.

Contrairement à beaucoup de livres techniques où il est courant de lire en diagonale et de sauter certains chapitres, il est ici très fortement recommandé de suivre l'ordre du cours, à moins que vous ne soyez déjà un peu expérimentés.

Pratiquez en même temps

Pratiquez régulièrement. N'attendez pas d'avoir fini de lire ce livre pour allumer votre ordinateur et faire vos propres essais.

Utilisez les codes web !

Afin de tirer parti du Site du Zéro dont ce livre est issu, celui-ci vous propose ce qu'on appelle des « codes web ». Ce sont des codes à six chiffres à saisir sur une page du Site du Zéro pour être automatiquement redirigé vers un site web sans avoir à en recopier l'adresse.

Pour utiliser les codes web, rendez-vous sur la page suivante :

<http://www.siteduzero.com/codeweb>

Un formulaire vous invite à rentrer votre code web. Faites un premier essai avec le code ci-dessous :

▷
Code web : 123456

Ces codes web ont deux intérêts :

- ils vous redirigent vers les sites web présentés tout au long du cours, vous permettant ainsi d'obtenir les logiciels dans leur toute dernière version ;
- ils vous permettent de télécharger les codes sources inclus dans ce livre, ce qui vous évitera d'avoir à recopier certains programmes un peu longs.

Ce système de redirection nous permet de tenir à jour le livre que vous avez entre les mains sans que vous ayez besoin d'acheter systématiquement chaque nouvelle édition. Si un site web change d'adresse, nous modifierons la redirection mais le code web à utiliser restera le même. Si un site web disparaît, nous vous redirigerons vers une page du Site du Zéro expliquant ce qui s'est passé et vous proposant une alternative.

En clair, c'est un moyen de nous assurer de la pérennité de cet ouvrage sans que vous ayez à faire quoi que ce soit !

Remerciements

Plus d'un an s'est écoulé entre l'écriture des premières lignes de ce cours et sa publication sous la forme d'un livre. Durant toute cette période, de nombreuses personnes ont participé plus ou moins directement à ce projet. C'est pourquoi je voudrais remercier :

- Ma famille, ma chérie et mes amis qui m'ont encouragés du début à la fin ;
- Fumble pour ses relectures, ses conseils et ses encouragements tout au long de la rédaction de ce cours ;
- Anna Schurtz pour ses conseils afin de rendre ce cours plus pédagogique ;
- L'ensemble de l'équipe de Simple IT ;
- Les membres du site OpenClassRooms pour leurs nombreux commentaires et encouragements.

Sommaire

Avant-propos	i
Qu'allez-vous apprendre en lisant ce livre?	ii
Comment lire ce livre?	ii
Remerciements	iii
I Les bases du XML	1
1 Qu'est-ce que le XML?	3
Qu'est ce que le XML?	4
Origine et objectif du XML	5
2 Les bons outils	7
L'éditeur de texte	8
EditiX	10
<oXygen/> XML Editor	12
3 Les éléments de base	15
Les balises	16
Les attributs	18
Les commentaires	18
4 Votre premier document XML	21
Structure d'un document XML	22

Un document bien formé	23
Utilisation d>EditiX	24
5 TP : structuration d'un répertoire	29
L'énoncé	30
Exemple de solution	30
II Créez des définitions pour vos documents XML	33
6 Introduction aux définitions et aux DTD	35
Qu'est-ce que la définition d'un document XML ?	36
Définition d'une DTD	37
Les éléments	37
Structurer le contenu des balises	40
7 DTD : les attributs et les entités	47
Les attributs	48
Les entités	53
8 DTD : où les écrire ?	57
Les DTD internes	58
Les DTD externes	59
Un exemple avec EditiX	62
9 TP : définition DTD d'un répertoire	67
L'énoncé	68
Une solution	69
10 Schéma XML : introduction	73
Les défauts des DTD	74
Les apports des schémas XML	74
Structure d'un schéma XML	75
Référencer un schéma XML	76
11 Schéma XML : les éléments simples	79
Les éléments simples	80

Les attributs	82
12 Schéma XML : les types simples	85
Les types chaînes de caractères	86
Les types dates	90
Les types numériques	94
Les autres types	98
13 Schéma XML : les types complexes	101
Définition	102
Les contenus simples	103
Les contenus "standards"	105
14 Schéma XML : aller plus loin	113
Le nombre d'occurrences	114
La réutilisation des éléments	115
L'héritage	123
Les identifiants	133
Un exemple avec EditiX	136
15 TP : Schéma XML d'un répertoire	141
L'énoncé	142
Une solution	143
III Traitez vos données XML	145
16 DOM : Introduction à l'API	147
Qu'est-ce que L'API DOM ?	148
L'arbre XML	148
Le vocabulaire et les principaux éléments	150
17 DOM : Exemple d'utilisation en Java	155
Lire un document XML	156
Ecrire un document XML	162
18 XPath : Introduction à l'API	169

SOMMAIRE

Qu'est-ce que l'API XPath ?	170
Chemin relatif et chemin absolu	172
19 XPath : Localiser les données	175
Dissection d'une étape	176
Les axes	176
Les tests de noeuds	177
Les prédictats	181
Un exemple avec EditiX	183
20 TP : des expressions XPath dans un répertoire	187
L'énoncé	188
Une solution	190
IV Transformez vos documents XML	191
21 Introduction à XSLT	193
Qu'est-ce que XSLT ?	194
Structure d'un document XSLT	195
Référencer un document XSLT	198
22 Les templates	201
Introduction aux templates	202
Contenu d'un template	202
Les fonctions	204
La fonction value-of	206
La fonction for-each	208
La fonction sort	209
La fonction if	211
La fonction choose	213
La fonction apply-templates	215
23 Les templates : aller plus loin	219
Les variables et la fonction call-template	220
Les variables	220

La fonction call-template	223
D'autres fonctions	228
Un exemple avec EditiX	232
24 TP : des transformations XSLT d'un répertoire	237
Le sujet	238
Une solution	240
V Annexes	241
25 Les espaces de noms	243
Définition	244
Utilisation d'un espace de noms	245
Quelques espaces de noms utilisés régulièrement	247
26 Mettez en forme vos documents XML avec CSS	249
Ecrire un document CSS	250
Un exemple avec EditiX	254
TP : mise en forme d'un répertoire	256

SOMMAIRE

Première partie

Les bases du XML

Chapitre

1

Qu'est-ce que le XML ?

Difficulté : 

Voici donc le tout premier chapitre de la longue série que va comporter ce tutoriel.

Je vous propose de tout d'abord faire connaissance avec cette technologie maintenant âgée de plus de 10 ans. Ce premier chapitre sera donc l'occasion de découvrir ce qu'est le **XML**, son origine et les besoins auxquels cette technologie devait répondre au moment de son élaboration.



Qu'est ce que le XML ?

Première définition

Le XML ou eXtensible Markup Language est un langage informatique de **balisage générique**.

Cette définition est à mes yeux quelque peu barbare et technique, c'est pourquoi, je vous propose d'en décortiquer les mots-clefs.

Un langage informatique

Vous n'êtes pas sans savoir qu'en informatique, il existe plusieurs centaines de langages destinés à des utilisations très diverses.

En vulgarisant un peu (beaucoup !), il est possible de les regrouper en 3 grandes catégories :

- Les **langages de programmation**.
- Les **langages de requête**.
- Les **langages de description**.

Les **langages de programmation** permettent de créer des programmes, des applications mobiles, des sites Internet, des systèmes d'exploitation, etc. Certains langages de programmation sont extrêmement populaires. En mai 2012, les 10 langages de programmation les plus populaires étaient le C, le Java, le C++, l'Objective-C, le C#, le PHP, le Basic, le Python, le Perl et le Javascript.

Les **langages de requêtes** permettent quant à eux d'interroger des structures qui contiennent des données. Parmi les langages de requête les plus connus, on peut par exemple citer le SQL pour les bases de données relationnelles, le SPARQL pour les graphes RDF et les ontologies OWL ou encore le XQuery pour les documents XML.

Enfin, les **langages de description** permettent de décrire et structurer un ensemble de données selon un jeu de règles et des contraintes définies. On peut par exemple utiliser ce type de langage pour décrire l'ensemble des livres d'une bibliothèque, ou encore la liste des chansons d'un CD, etc. Parmi les langages de description les plus connus, on peut citer le SGML, le XML ou encore le HTML.

Un langage de balisage générique

Un **langage de balisage** est un langage qui s'écrit grâce à des **balises**. Ces balises permettent de structurer de manière hiérarchisée et organisée les données d'un document.

Si vous ne savez pas ce qu'est une **balise**, ne vous inquiétez pas, nous reviendrons sur ce terme un peu plus loin dans le cours.

Le terme **générique** signifie que nous allons pouvoir créer nos propres balises. Nous ne sommes pas obligés d'utiliser un ensemble de balises existantes comme c'est par

exemple le cas en HTML.

Une nouvelle définition

Après les explications que nous venons de voir ensemble, je vous propose une nouvelle définition du langage XML bien moins technique que la première donnée au début de ce cours.

Voici donc cette nouvelle définition : *le langage XML est un langage qui permet de décrire des données à l'aide de balises et de règles que l'on peut personnaliser.*

Certaines notions peuvent encore vous sembler abstraites, mais ne vous inquiétez pas, tout sera expliqué dans la suite de ce tutoriel.

Origine et objectif du XML

La petite histoire du XML

Vous vous doutez bien que le **langage XML** n'a pas été créé dans le simple but de s'amuser. Sa création avait pour objectif de répondre à un besoin très précis : **l'échange de données**.

Aux débuts d'Internet, les ordinateurs et les programmes échangeaient des données en utilisant des fichiers. Malheureusement, ces fichiers avaient bien souvent des règles de formatage qui leur étaient propres. Par exemple, les données étaient séparées par des points, des virgules, des espaces, des tirets, etc.

Le problème avec ce système est qu'il fallait sans cesse adapter les programmes au format du fichier ce qui représentait une charge de travail importante. Il a donc fallu régler ce problème rapidement. Le langage **SGML** ou Standard Generalized Markup Language est alors né. C'est un langage puissant, extensible et standard qui permet de décrire à l'aide de **balises** un ensemble de données. Mais ce langage, très complexe, n'était pas forcément compatible pour effectuer des échanges sur le web.

Un groupe d'informaticiens ayant de l'expérience dans le SGML et le web a alors décidé de se pencher sur le sujet. Le langage **XML** est donc né. Le XML 1.0 est devenu une recommandation du W3C (le « *World Wide Web Consortium* ») le 10 février 1998.

Depuis, les spécifications du langage ont évolué, et la version 1.1 a été publiée le 4 février 2004. C'est pourtant la version 1.0 du langage XML qui est encore la plus utilisée aujourd'hui et c'est cette version que nous allons étudier dans ce tutoriel.

Les objectifs du XML

Comme nous l'avons vu, l'objectif du XML est de faciliter les échanges de données entre les machines. A cela s'ajoute un autre objectif important : **décrire les données de manière aussi bien compréhensible par les hommes qui écrivent les**

documents XML que par les machines qui les exploitent.

Le XML se veut également compatible avec le web afin que les échanges de données puissent se faire facilement à travers le réseau Internet.

Le XML se veut donc standardisé, simple, mais surtout extensible et configurable afin que n'importe quel type de données puisse être décrit.

En résumé

- Le XML a été créé pour faciliter les échanges de données entre les machines et les logiciels.
- Le XML est un langage qui s'écrit à l'aide de **balises**.
- Le XML est une recommandation du **W3C**, il s'agit donc d'une technologie avec des règles strictes à respecter.
- Le XML se veut compréhensible par tous : les hommes comme les machines.
- Le XML nous permet de créer notre propre vocabulaire grâce à un ensemble de règles et de balises personnalisables.

Chapitre 2

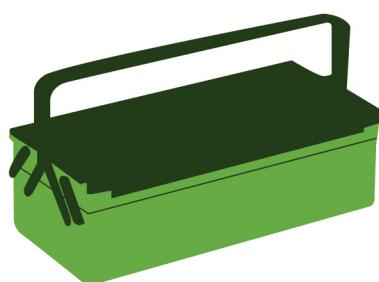
Les bons outils

Difficulté : 

Après ce premier chapitre, peu technique, destiné à vous introduire le **XML**, j'espère que l'utilité de ce dernier est plus claire dans vos esprits. Dans ce second chapitre, nous allons passer en revue les outils et tenter de comprendre les logiciels qui nous seront utiles pour manipuler les différentes technologies que nous verrons tout au long de ce cours.

L'utilisation d'outils spécialisés va nous faciliter la vie et nous permettre d'être beaucoup plus productifs. En effet, tous proposent des fonctions clefs destinées à nous aider comme par exemple la **coloration syntaxique** ou encore la **validation de documents**.

Je vous encourage à essayer vous-mêmes les différents logiciels qui seront présentés dans ce chapitre et d'adopter celui qui vous correspond le plus.



L'éditeur de texte

Il faut savoir qu'un **document XML** n'est en réalité qu'un simple document texte. C'est pourquoi, il est tout à fait possible d'utiliser un éditeur de texte pour la rédaction de nos documents XML.

Sous Windows

Sous Windows, un éditeur de texte portant le nom de **Bloc-notes** est généralement installé par défaut. En théorie, il est suffisant et fonctionnel. Dans la pratique, les fonctionnalités qu'il offre sont limitées et des options très utiles comme la coloration syntaxique ou la numérotation des lignes manquent.

Je vous propose donc d'utiliser **Notepad++** qui est parfait pour ce que nous souhaitons réaliser puisqu'il pallie idéalement les manques du Bloc-notes. Il s'agit d'un logiciel gratuit, n'hésitez donc pas à le télécharger !

▷ Télécharger Notepad++
Code web : 480303

Je ne vais pas détailler ici la procédure d'installation qui est classique pour un logiciel tournant sous Windows.

Une fois installé, lancez le logiciel. Vous devriez avoir une fenêtre semblable à la figure 2.1.

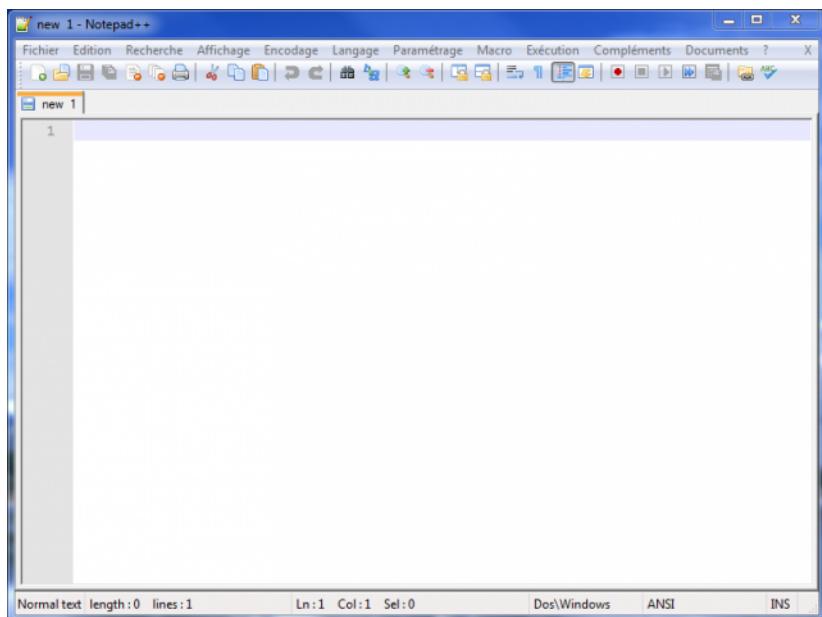


FIGURE 2.1 – Notepad++

Afin d'adapter la coloration syntaxique au langage XML, il vous suffit de sélectionner **Langage** dans la barre de menu puis **XML** dans la liste.

Lorsque vous enregistrerez vos documents, il suffira alors de préciser comme extension ".xml" pour conserver la coloration syntaxique d'une fois sur l'autre.

Sous GNU/Linux

Par défaut, les distributions Linux sont souvent livrées avec de très bons éditeurs de texte. Si vous aimez la console, vous pouvez par exemple utiliser **nano**, **emacs**, **vi** ou encore **vim**. Si vous préférez les interfaces graphiques, je vous conseille d'utiliser l'excellent **gedit** qui normalement doit être installé par défaut. Si jamais ce n'est pas le cas, la commande suivante vous permettra de l'installer en quelques instants :

```
sudo apt-get install gedit
```

Une fois ouvert, vous devriez voir apparaître quelque chose ressemblant à la figure 2.2.

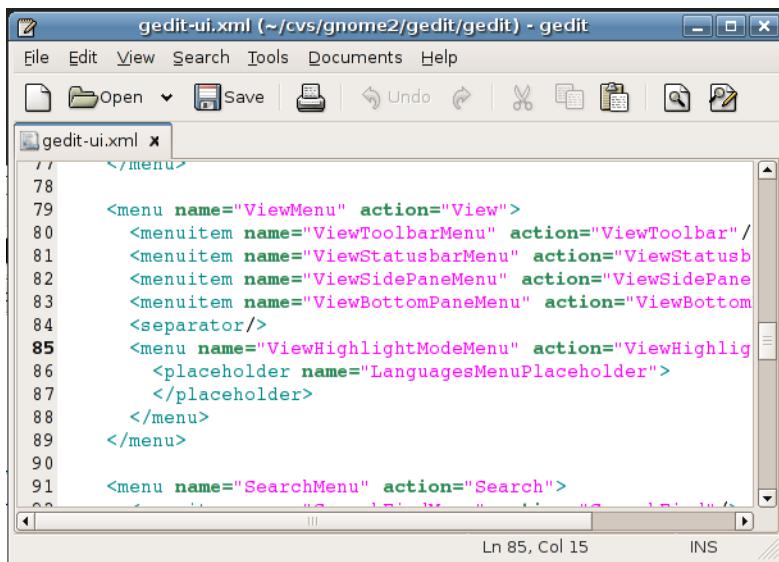


FIGURE 2.2 – gedit - photo issue du site officiel du projet

Afin d'adapter la coloration syntaxique au langage XML, il vous suffit de sélectionner **Affichage** dans la barre de menu puis **Mode de coloration** et finalement de choisir le **XML** dans la liste.

Lorsque vous enregistrerez vos documents, il suffira alors de préciser comme extension ".xml" pour conserver la coloration syntaxique une fois sur l'autre.

Sous MAC OS X

Pour les utilisateurs du système d'exploitation d'Apple, je vous conseille de vous tourner vers **jEdit**.

▷ Télécharger jEdit
Code web : 866884

La figure 2.3 vous donne un aperçu de son interface.

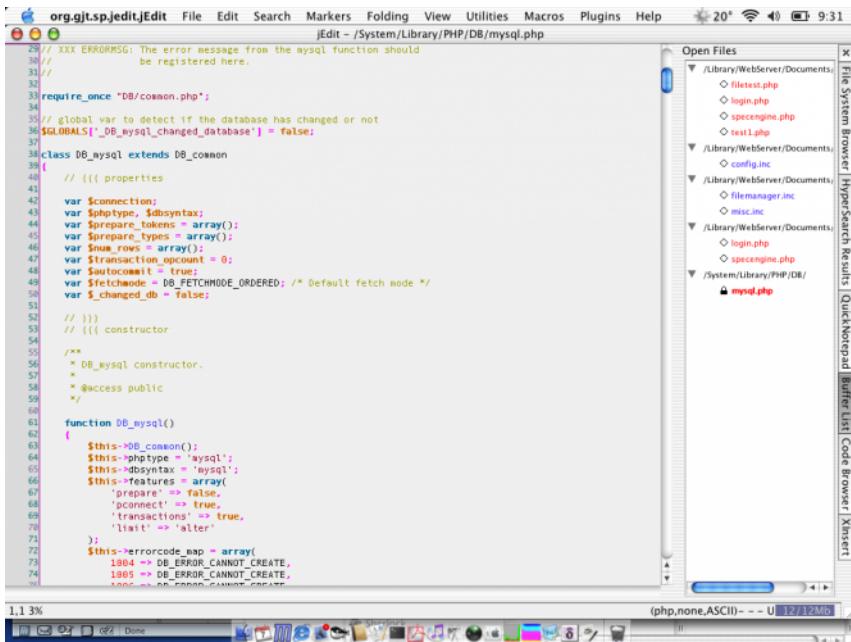


FIGURE 2.3 – jEdit - photo issue du site officiel du projet

EditiX

EditiX est un éditeur XML qui fonctionne sur les plateformes Windows, GNU/Linux ou Mac OS X. En plus de la **coloration syntaxique** essentielle à l'écriture de documents XML, ce logiciel nous offre de nombreux outils qui nous seront utiles dans la suite de ce tutoriel comme par exemple la **validation des documents**.

La version payante

Il existe plusieurs versions de ce logiciel. La dernière en date est **EditiX 2013**. Cette version complète est payante, mais plusieurs licences sont disponibles. Les étudiants peuvent par exemple bénéficier de 50% de réduction.

▷ Télécharger EditiX 2013
Code web : [264135](#)

La version gratuite

Heureusement, pour les pauvres Zéros fauchés que nous sommes, une version gratuite existe ! Il s'agit d'une version allégée d'**Editix 2008**. Notez bien que cette version est réservée à un usage non-commercial.

▷ Télécharger EditiX 2008, Lite
Version
Code web : [265354](#)

Puisque c'est cette version que je vais en partie utiliser dans la suite du tutoriel, je vous propose de détailler la procédure de mise en place du logiciel sous GNU/Linux. Je ne détaille pas la procédure d'installation sous Windows et MAC OS X puisqu'elle est des plus classique.

La mise en place sous GNU/Linux

Téléchargement et installation

Enregistrez l'archive sur votre bureau puis lancez votre plus beau terminal afin de débuter la procédure. Commencez par déplacer l'archive `editix-free-2008.tar.gz` fraîchement téléchargée dans le répertoire `/opt/` par la commande :

```
sudo mv ~/Bureau/editix-free-2008.tar.gz /opt/
```

Déplacez vous maintenant dans le dossier `/opt/` par la commande :

```
cd /opt/
```

Nous allons maintenant extraire les fichiers de l'archive que nous avons téléchargée. Pour ce faire, vous pouvez utiliser la commande :

```
sudo tar xvzf editix-free-2008.tar.gz
```

Un dossier nommé `editix` doit alors apparaître. Il contient les fichiers que nous venons d'extraire. Vous pouvez alors supprimer l'archive via la commande :

```
sudo rm editix-free-2008.tar.gz
```

On pourrait choisir de s'arrêter là et lancer le logiciel en ligne de commande en se rendant dans le répertoire `/opt/editix/bin/` et en exécutant le script `run.sh` via la commande :

```
./run.sh
```

Pour plus de confort, je vous propose plutôt de créer un **launcher**.

Création du launcher

Pour ce faire, faites un clic droit sur le menu **Applications** de votre tableau de bord (ou sur le **Menu** si vous êtes sur une LMDE par exemple) et cliquez sur **Editer le menu**.

Dans la colonne de gauche, choisissez le menu **Programmation** puis cliquez sur le bouton **Nouvel élément** dans la colonne de droite.

Une fenêtre devrait alors s'afficher (voir figure 2.4).

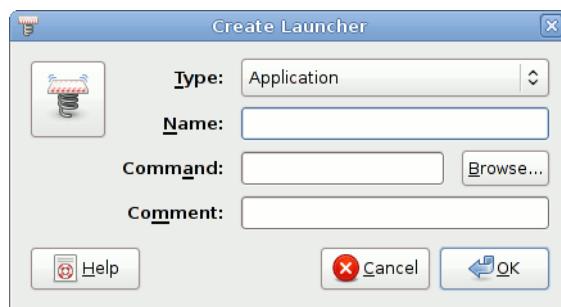


FIGURE 2.4 – Crétation du launcher

Remplissez le formulaire avec les informations suivantes :

- Type : Application
- Nom : Editix
- Commande : /opt/editix/bin/run.sh

Finalisez la création du launcher en cliquant sur le bouton **Valider**. Editix devrait maintenant apparaître dans vos applications et, plus particulièrement, dans le menu programmation.

Quel que soit votre système d'exploitation, la fenêtre du logiciel ressemble à la figure 2.5 après son lancement.

<oXygen/> XML Editor

Concluons cette présentation avec le logiciel **<oXygen/> XML Editor** (voir figure 2.6) qui, comme Editix, est multiplateformes. Contrairement à EditiX, il n'existe pas de version gratuite du logiciel, mais il reste possible de le tester gratuitement pendant 30 jours. Comme pour **Editix**, **<oXygen/> XML Editor** propose plusieurs types de licences. Ainsi, les étudiants peuvent obtenir des réductions très intéressantes.

▷ Télécharger <oXygen/>
Code web : 685988

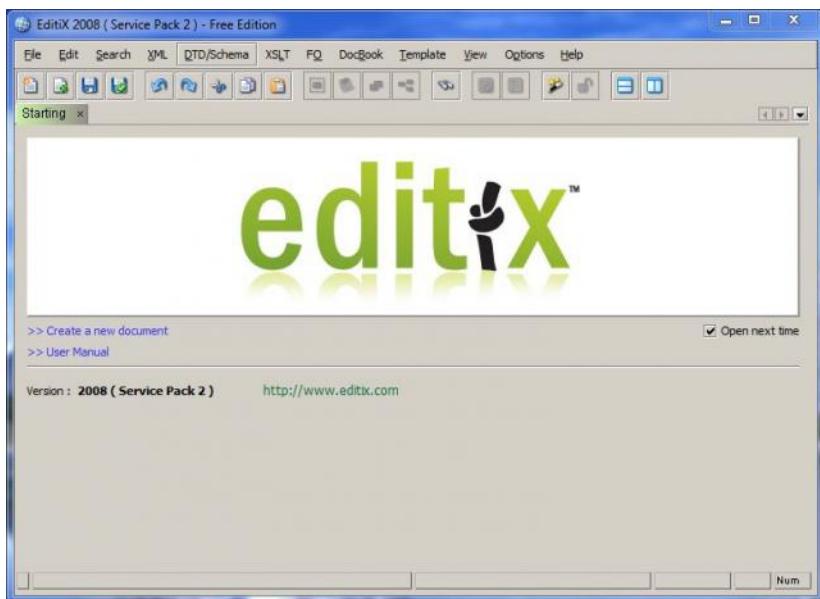


FIGURE 2.5 – Page de démarrage de Editix 2008 Lite Version

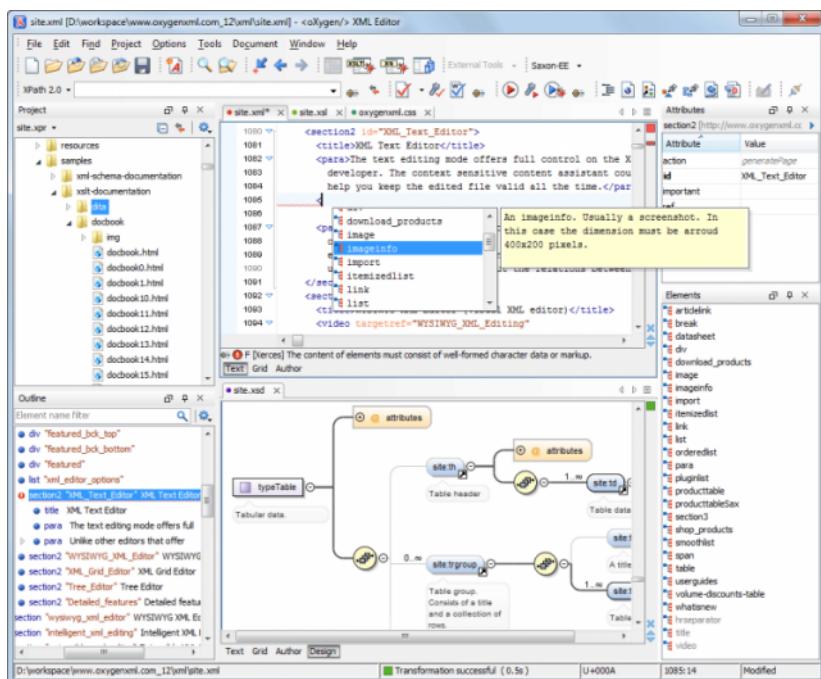


FIGURE 2.6 – <oXygen/> - photo issue du site officiel du projet

En résumé

- Même si ce n'est pas obligatoire, un logiciel spécialisé est conseillé.
- L'utilisation de **EditiX** sera abordée régulièrement dans ce tutoriel.

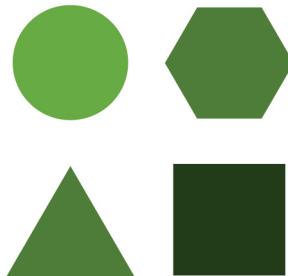
Chapitre 3

Les éléments de base

Difficulté : 

Avec les 2 précédents chapitres, nous ne sommes pas encore réellement entrés dans le vif du sujet, ils étaient simplement destinés à préparer le terrain.

Maintenant, les choses sérieuses commencent, nous allons entrer dans la partie pratique. Dans ce chapitre, nous allons découvrir ensemble les bases du XML. Après l'avoir lu, les mots clefs « **balise** » et « **attribut** » n'auront plus de secrets pour vous !



Les balises

Dans le tout premier chapitre, je définissais le langage XML comme un langage informatique de balisage. En effet, les **balises** sont les éléments de base d'un document XML. Une balise porte un nom qui est entouré de **chevrons**. Une balise commence donc par un < et se termine par un >. Par exemple : <balise> définit une balise qui s'appelle « **balise** ».

En XML, on distingue 2 types de balises : les **balises par paires** et les **balises uniques**.

Les balises par paires

Définition

Les **balises par paires** sont composées en réalité de 2 balises que l'on appelle **ouvrantes** et **fermantes**. La **balise ouvrante** commence par < et se termine par > tandis que la **balise fermante** commence par </ et se termine par >. Par exemple :

```
1 | <balise></balise>
```



Il est extrêmement important que les balises ouvrantes et fermantes aient exactement le même nom. XML est sensible à la casse (c'est-à-dire qu'il fait la distinction entre les majuscules et les minuscules)! *Toute balise ouverte doit impérativement être fermée. C'est une règle d'or!*

Bien évidemment, on peut mettre « des choses » entre ces balises. On parle alors de **contenu**. Par exemple :

```
1 | <balise>Je suis le contenu de la balise</balise>
```

Quelques règles

Une **balise par paires** ne peut pas contenir n'importe quoi : elle peut contenir une **valeur simple** comme par exemple une chaîne de caractères, un nombre entier, un nombre décimal, etc.

```
1 | <balise1>Ceci est une chaîne de caractères</balise1>
2 | <balise2>10</balise2>
3 | <balise3>7.5</balise3>
```

Une **balise par paires** peut également contenir une **autre balise**. On parle alors d'**arborescence**.

```
1 | <balise1>
2 |   <balise2>10</balise2>
3 | </balise1>
```



Faites cependant très attention, si une balise peut en contenir une autre, il est cependant interdit de les chevaucher. L'exemple suivant n'est pas du XML !

```
1 | <balise1><balise2>Ceci est une chaîne de caractères</balise1></
   balise2>
```

Enfin, une **balise par paires** peut contenir un **mélange de valeurs simples et de balises** comme en témoigne l'exemple suivant :

```
1 | <balise1>
2 |   Ceci est une chaîne de caractères
3 |   <balise2>10</balise2>
4 |   7.5
5 | </balise1>
```

Les balises uniques

Une **balise unique** est en réalité une balise par paires qui n'a pas de contenu. Vous le savez, les informaticiens sont des fainéants ! Ainsi, plutôt que de perdre du temps à ouvrir et fermer des balises sans rien écrire entre, une syntaxe un peu spéciale a été mise au point :

```
1 | <balise />
```

Les règles de nommage des balises

Ce qui rend le XML générique, c'est la possibilité de créer votre propre langage balisé. Ce **langage balisé**, comme son nom l'indique, est un langage composé de balises sauf qu'en XML, c'est vous qui choisissez leurs noms.

L'exemple le plus connu des langages balisés de type XML est très certainement le xHTML qui est utilisé dans la création de sites Internet.

Il y a cependant quelques règles de nommage à respecter pour les balises de votre langage balisé :

- Les noms peuvent contenir des lettres, des chiffres ou des caractères spéciaux.
- Les noms ne peuvent pas débuter par un nombre ou un caractère de ponctuation.
- Les noms ne peuvent pas commencer par les lettres XML (quelle que soit la casse).
- Les noms ne peuvent pas contenir d'espaces.
- On évitera les caractères - , ; . < et > qui peuvent être mal interprétés dans vos programmes.

Les attributs

Définition

Il est possible d'ajouter à nos balises ce qu'on appelle des **attributs**. Tout comme pour les balises, c'est vous qui en choisissez le nom. Un **attribut** peut se décrire comme une option ou une donnée cachée. Ce n'est pas l'information principale que souhaite transmettre la balise, mais il donne des renseignements supplémentaires sur son contenu.

Pour que ce soit un peu plus parlant, voici tout de suite un exemple :

```
1 | <prix devise="euro">25.3</prix>
```

Dans l'exemple ci-dessus, l'information principale est le prix. L'attribut `devise` nous permet d'apporter des informations supplémentaires sur ce prix, mais ce n'est pas l'information principale que souhaite transmettre la balise `<prix>`.

Une balise peut contenir 0 ou plusieurs attributs. Par exemple :

```
1 | <prix devise="euro" moyen_paiement="chèque">25.3</prix>
```

Quelques règles

Tout comme pour les balises, quelques règles sont à respecter pour les attributs :

- Les règles de nommage sont les mêmes que pour les balises.
- La valeur d'un attribut doit impérativement être délimitée par des guillemets, simples ou doubles.
- Dans une balise, un attribut ne peut-être présent qu'une seule fois.

Les commentaires

Avant de passer à la création de notre premier document XML, j'aimerais vous parler des **commentaires**.

Un **commentaire** est un texte qui permet de donner une indication sur ce que l'on fait. Il vous permet d'annoter votre fichier et d'expliquer une partie de celui-ci.

En XML, les commentaires ont une syntaxe particulière. C'est une balise unique qui commence par `<!--` et qui se termine par `-->`.

```
1 | <!-- Ceci est un commentaire ! -->
```

Voyons tout de suite sur un exemple concret :

```
1 | <!-- Description du prix -->
2 | <prix devise="euro">12.5</prix>
```

Certes, sur cet exemple, les commentaires semblent un peu inutiles, mais je vous assure qu'ils vous seront d'une grande aide pendant la rédaction de longs documents XML !

En résumé

- Deux types de balises existent : les **balises par paires** et les **balises uniques**.
- Les balises peuvent contenir des **attributs**.
- Un document XML peut contenir des **commentaires**.

Votre premier document XML

Difficulté : 

Dans le chapitre précédent, nous avons découvert les éléments de base du XML, mais vous ignorez encore comment écrire un document XML. Ne vous inquiétez pas, ce chapitre a pour objectif de corriger ce manque.

Dans ce chapitre, nous allons donc nous attaquer à tout ce qui se rattache à l'écriture d'un **document XML**. Ce sera l'occasion de découvrir la structure générale d'un document ainsi que nouvelles notions clefs. Nous finirons par nous lancer dans la pratique, seulement un petit peu, rassurez-vous, à travers l'utilisation du logiciel **EditiX** afin d'écrire notre premier document XML en bonne et due forme !



Structure d'un document XML

Un document XML peut être découpé en 2 parties : le **prologue** et le **corps**.

Le prologue

Le **prologue** correspond à la première ligne de votre document XML. Il donne des informations de traitement.

Voici à quoi notre prologue ressemble dans cette première partie du tutoriel :

```
1 | <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
```

Comme vous pouvez le remarquer, le prologue est une balise unique qui commence par `<?xml` et qui se termine par `?>`. Si vous ne comprenez pas cette ligne, pas de panique, nous allons tout décortiquer ensemble pas à pas.

La version

Dans le prologue, on commence généralement par indiquer la version de XML que l'on utilise pour décrire nos données. Pour rappel, il existe actuellement 2 versions : 1.0 et 1.1.

À noter : le prologue n'est obligatoire que depuis la version 1.1, mais il est plus que conseillé de l'ajouter quand même lorsque vous utilisez la version 1.0.

La différence entre les 2 versions est une amélioration dans le support des différentes versions de l'Unicode. Sauf si vous souhaitez utiliser des caractères chinois dans vos documents XML, il conviendra d'utiliser la version 1.0 qui est encore aujourd'hui la version la plus utilisée.

Le jeu de caractères

La seconde information de mon prologue est `encoding="UTF-8"`.

Il s'agit du **jeu de caractères** utilisé dans mon document XML. Par défaut, l'encodage de XML est l'UTF-8, mais si votre éditeur de texte enregistre vos documents en ISO8859-1, il suffit de la changer dans le prologue :

```
1 | <?xml version = "1.0" encoding="ISO8859-1" standalone="yes" ?>
```

Un document autonome

La dernière information présente dans le prologue est `standalone="yes"`.

Cette information permet de savoir si votre document XML est autonome ou si un autre document lui est rattaché.

Il est encore un peu tôt pour vous en dire plus. Nous reviendrons sur cette notion dans la partie 2 du tutoriel. Pour le moment, acceptez le fait que nos documents sont tous autonomes.

Le corps

Le **corps** d'un document XML est constitué de l'ensemble des balises qui décrivent les données. Il y a cependant une règle très importante à respecter dans la constitution du corps : *une balise en paires unique doit contenir toutes les autres*. Cette balise est appelée **élément racine** du corps.

Voyons tout de suite un exemple :

```
1 <racine>
2   <balise_paire>texte</balise_paire>
3   <balise_paire2>texte</balise_paire2>
4   <balise_paire>texte</balise_paire>
5 </racine>
```

Bien évidemment, lorsque vous créez vos documents XML, le but est d'être le plus explicite possible dans le nommage de vos balises. Ainsi, le plus souvent, la **balise racine** aura pour mission de décrire ce quelle contient.

Si je choisis de décrire un répertoire, je peux par exemple nommer mes balises comme dans l'exemple suivant :

```
1 <repertoire>
2   <personne>Bernard</personne>
3   <personne>Patrick</personne>
4 </repertoire>
```

Un document complet

Un document XML, certes simple, mais complet pourrait donc être le suivant :

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <personne>Bernard</personne>
4   <personne>Patrick</personne>
5 </repertoire>
```

Un document bien formé

Quand vous entendrez parler de XML, vous entendrez souvent parler de **document XML bien formé** ou **well-formed** en anglais.

Cette notion décrit en réalité un document XML *conforme aux règles syntaxiques* décrites tout au long de cette première partie du tutoriel.

On peut résumer un document XML bien formé à un document XML avec une syntaxe correcte, c'est-à-dire :

- S'il s'agit d'un document utilisant la version 1.1 du XML, le prologue est bien renseigné.
- Le document XML ne possède qu'une seule balise racine.
- Le nom des balises et des attributs est conforme aux règles de nommage.
- Toutes les balises en paires sont correctement fermées.
- Toutes les valeurs des attributs sont entre guillemets simples ou doubles.
- Les balises de votre document XML ne se chevauchent pas, il existe une arborescence dans votre document.

Si votre document XML est bien formé, félicitation, il est exploitable ! Dans le cas contraire, votre document est inutilisable.

Utilisation d'EditiX

Tout cela est bien beau, me direz-vous, mais depuis le début de ce tutoriel, nous parlons du XML sans avoir encore utilisé les logiciels du chapitre 2.

Nous allons les employer dès maintenant. Dans cette partie, nous allons créer notre premier document XML grâce au logiciel **EditiX** et vérifier qu'il est bien formé.

Je vous propose d'utiliser le document complet que nous avons construit auparavant :

```
1 | <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 | <repertoire>
3 |   <personne>Robert</personne>
4 |   <personne>John</personne>
5 | </repertoire>
```

Créer un nouveau document

Commencez par lancer EditiX.

Pour créer un nouveau document, vous pouvez cliquer sur l'icône présentée à la figure 8.2 puis sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier **Ctrl** + **N**.



FIGURE 4.1 – Nouveau document

Dans la liste qui s'affiche, sélectionnez **Standard XML document**, comme indiqué sur à la figure 4.2.

Surprise ! Votre document XML n'est pas vierge. Voici ce que vous devriez voir :

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

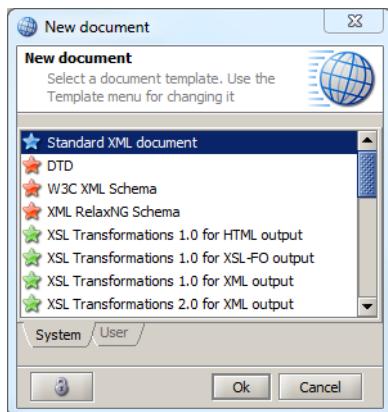


FIGURE 4.2 – Création d'un document XML

```

2 | 
3 | <!-- New document created with EditiX at Fri May 18 00:11:02
   | CEST 2012 -->

```

Comme vous pouvez le constater, EditiX s'est chargé pour vous d'écrire le prologue de votre document XML. Il a également pris en charge la rédaction d'un petit commentaire pour vous rappeler la date et l'heure de création de votre document.

Puisque notre document sera autonome, vous pouvez modifier le prologue pour l'indiquer :

```
1 | <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

Vérification du document

Nous pouvons vérifier dès maintenant si notre document est bien formé. Pour ce faire, vous pouvez cliquer sur l'icône représentée à la figure 8.4 et sélectionner dans la barre de menu **XML** puis **Check this document** ou encore utiliser le raccourci clavier **Ctrl** + **K**.



FIGURE 4.3 – Vérification d'un document

Vous devriez alors voir une erreur s'afficher. La ligne où se situe l'erreur est représentée par un rectangle aux bords rouges sur notre espace de travail, comme indiqué sur la figure 4.4.

Nous avons donc une erreur à la ligne 6 de notre document. Pour en savoir plus sur notre erreur, il suffit de regarder en bas de l'écran. La figure 4.5 montre ce que vous devriez voir.



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2
3 <!-- New document created with Editix at Fri May 18 00:11:02 CEST 2012 -->
4
5
6
7
8
9
```

FIGURE 4.4 – Le document XML contient une erreur

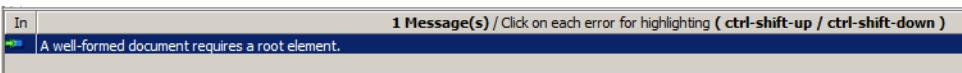


FIGURE 4.5 – Message d’erreur indiquant que le document n’est pas bien formé

Pour ceux qui ne parlent pas anglais, voici ce que dit le message : « *Un document bien formé nécessite un élément racine.* »

Il manque donc un **élément racine**. Complétons tout de suite notre document avec les éléments suivant :

```
1 <repertoire>
2   <personne>Robert</personne>
3   <personne>John</personne>
4 </repertoire>
```

Lancez de nouveau la vérification du document. Vous devriez avoir le message présent à la figure 26.4 à l’écran.

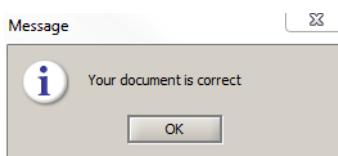


FIGURE 4.6 – Message indiquant un document XML bien formé

Félicitation, votre document est bien formé !

L’indentation

Il est possible de demander à Editix d'**indenter automatiquement** vos documents une fois écrits. Pour ce faire, sélectionnez dans la barre de menu **XML** puis **Format et Pretty format (default)** ou utilisez le raccourci clavier **Ctrl + R**.

Dans ce même menu, vous pouvez accéder aux paramètres concernant la tabulation.

L’arborescence du document

Editix met à votre disposition un outil fort sympathique qui vous permet de visualiser l’arborescence du document en cours d’édition (voir figure 4.7).

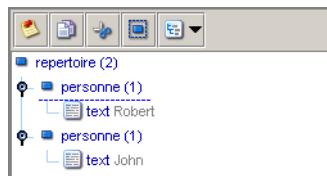


FIGURE 4.7 – Arborescence du document XML

On sait ainsi que notre répertoire contient 2 personnes : Robert et John.

Enregistrer votre document

Il est maintenant temps de clore ce chapitre en enregistrant votre document XML. Pour ce faire, vous pouvez cliquer sur l’icône visible à la figure 4.8 ou bien sélectionner dans la barre de menu **File** puis **Save** ou encore utiliser le raccourci clavier **Ctrl** + **S**.



FIGURE 4.8 – Enregistrement d’un document

Dans la fenêtre de dialogue qui vient de s’ouvrir, choisissez l’emplacement dans lequel vous souhaitez stocker votre fichier XML, tapez son nom et cliquez sur **Enregistrer**.

En résumé

- Un document XML est composé de 2 parties : le **prologue** et le **corps**.
- un document XML doit être bien formé pour être exploitable.
- **EditiX** permet de vérifier qu’un document est bien formé en seulement quelques clics.

Chapitre 5

TP : structuration d'un répertoire

Difficulté : 

Voici donc le premier TP de ce tutoriel ! L'objectif de ces chapitres TP, un peu particuliers, est de vous inviter à vous lancer dans la pratique à l'aide de tous les éléments théoriques que vous avez lu au cours des chapitres précédents. Cela me semble indispensable pour s'assurer que vous avez bien compris toutes les notions abordées jusqu'à maintenant.

Dans ce premier TP, l'objectif est de vous montrer une utilisation concrète de structuration de données via XML.



L'énoncé

Le but de ce TP est de créer un document XML structurant les données d'un répertoire. Votre répertoire doit comprendre au moins 2 personnes. Pour chaque personne, on souhaite connaître les informations suivantes :

- Son sexe (homme ou femme).
- Son nom.
- Son prénom.
- Son adresse.
- Un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.).
- Une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Je ne vous donne aucune indication concernant le choix des balises, des attributs et de l'arborescence à choisir pour une raison très simple : lorsque l'on débute en XML, le choix des attributs, des balises et de l'arborescence est assez difficile.

L'objectif est vraiment de vous laisser chercher et vous pousser à vous poser les bonnes questions sur l'utilité d'une balise, d'un attribut, etc.

Exemple de solution

Je vous fais part de ma solution. Notez bien que ce n'est qu'une solution parmi les multiples solutions possibles !

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3  <repertoire>
4      <!-- John DOE -->
5      <personne sexe="masculin">
6          <nom>DOE</nom>
7          <prenom>John</prenom>
8          <adresse>
9              <numero>7</numero>
10             <voie type="impasse">impasse du chemin</voie>
11             <codePostal>75015</codePostal>
12             <ville>PARIS</ville>
13             <pays>FRANCE</pays>
14         </adresse>
15         <telephones>
16             <telephone type="fixe">01 02 03 04 05</telephone>
17             <telephone type="portable">06 07 08 09 10</
18                 telephone>
19         </telephones>
20         <emails>
21             <email type="personnel">john.doe@wanadoo.fr</email>
22             <email type="professionnel">john.doe@societe.com</
23                 email>
24         </emails>
```

```
23    </personne>
24
25    <!-- Marie POPPINS -->
26    <personne sexe="feminin">
27        <nom>POPPINS</nom>
28        <prenom>Marie</prenom>
29        <adresse>
30            <numero>28</numero>
31            <voie type="avenue">avenue de la république</voie>
32            <codePostal>13005</codePostal>
33            <ville>MARSEILLE</ville>
34            <pays>FRANCE</pays>
35        </adresse>
36        <telephones>
37            <telephone type="professionnel">04 05 06 07 08</
38                telephone>
39        </telephones>
40        <emails>
41            <email type="professionnel">contact@poppins.fr</
42                email>
43        </emails>
44    </personne>
45 </repertoire>
```

Quelques explications

Le sexe

Comme vous pouvez le constater, j'ai fait le choix de renseigner le sexe dans un attribut de la balise `<personne/>` et non d'en faire une balise à part entière.

En effet, cette information est, je pense, plus utile à l'ordinateur qui lira le document qu'à toute personne qui prendrait connaissance de ce fichier. En effet, contrairement à une machine, nous avons la capacité de déduire que John est un prénom masculin et Marie, un prénom féminin. Cette information n'est donc pas cruciale pour les personnes qui lisent le fichier.

L'adresse

Il est important que vos documents XML aient une arborescence *logique*. C'est pourquoi j'ai décidé de représenter l'adresse postale par une balise `<adresse />` qui contient les informations détaillées de l'adresse de la personne comme le numéro de l'immeuble, la voie, le pays, etc.

J'ai également fait le choix d'ajouter un attribut `type` dans la balise `<voie />`. Une nouvelle fois, cet attribut est destiné à être utilisé par une machine.

En effet, une machine qui traitera ce fichier, pourra facilement accéder au type de la voie sans avoir à récupérer le contenu de la balise `<voie/>` et tenter d'analyser s'il

s'agit d'une impasse, d'une rue, d'une avenue, etc. C'est donc un gain de temps dans le traitement des données.

Numéros de téléphone et adresses e-mails

Encore une fois, dans un soucis d'arborescence logique, j'ai décidé de créer les blocs <telephones /> et <emails /> qui contiennent respectivement l'ensemble des numéros de téléphone et des adresses e-mail.

Pour chacune des balises <telephone/> et <email/>, j'ai décidé d'y mettre un attribut **type**. Cet attribut permet de renseigner si l'adresse e-mail ou le numéro de téléphone est par exemple professionnel ou personnel.

Bien qu'indispensable aussi bien aux êtres humains qu'aux machines, cette information est placée dans un attribut car ce n'est pas l'information principale que l'on souhaite transmettre. Ici, l'information principale reste le numéro de téléphone ou l'adresse e-mail et non son type.

Deuxième partie

Créez des définitions pour vos documents XML

Introduction aux définitions et aux DTD

Difficulté : 

Dans la première partie, nous avons découvert ce qu'était le XML. Pour ne rien vous cacher, le XML est très peu utilisé seul et on préfère souvent l'utiliser en parallèle d'autres technologies permettant d'étendre les possibilités de la technologie. L'étude de ces autres technologies débute donc maintenant ! Nous allons aborder les technologies qui permettent de définir une structure stricte aux documents XML : les **fichiers de définition**.

Il existe plusieurs technologies qui permettent d'écrire des fichiers de configuration, nous en verrons 2 dans cette seconde partie. Débutons tout de suite avec les **DTD**.



Qu'est-ce que la définition d'un document XML ?

Avant de foncer tête baissée dans la seconde partie de ce cours, il est indispensable de revenir sur quelques termes qui seront importants pour la suite de ce tutoriel.

Quelques définitions

Définition d'une définition

Une **définition d'un document XML** est un ensemble de règles que l'on impose au document. Ces règles permettent de décrire la façon dont le document XML doit être construit. Elles peuvent être de natures différentes. Par exemple, ces règles peuvent imposer la présence d'un attribut ou d'une balise, imposer l'ordre d'apparition des balises dans le document ou encore, imposer le type d'une donnée (nombre entier, chaîne de caractères, etc.).

Un document valide

Dans la partie précédente, nous avons vu ce qu'était un document **bien formé**. Cette seconde partie est l'occasion d'aller un peu plus loin et de voir le concept de document **valide**.

Un document valide est un document bien formé conforme à une définition. Cela signifie que le document XML respecte toutes les règles qui lui sont imposées dans les fameuses définitions.

Pourquoi écrire des définitions ?

Vous vous demandez certainement à quoi servent ces définitions et pourquoi on les utilise, n'est-ce pas ?

Associer une définition à un document oblige à une certaine rigueur dans l'écriture de vos données XML. C'est d'autant plus important lorsque plusieurs personnes travaillent sur un même document. La définition impose ainsi une écriture uniforme que tout le monde doit respecter. On évite ainsi que l'écriture d'un document soit anarchique et, par conséquent, difficilement exploitable.



Explicable oui ! Mais par qui ?

Le plus souvent, par un programme informatique ! Vous pouvez par exemple écrire un programme informatique qui traite les données contenues dans un document XML respectant une définition donnée. Imposer une définition aux documents que votre programme exploite permet d'assurer un automatisme et un gain de temps précieux :

- **Le document n'est pas valide** : je ne tente pas de l'exploiter.
- **Le document est valide** : je sais comment l'exploiter.

Pour terminer cette longue introduction, sachez que vous avez le choix entre deux technologies pour écrire les définitions de vos documents XML : les **DTD** ou les **schémas XML**.

Définition d'une DTD

Une définition rapide

Une **Document Type Definition** ou en français une **Définition de Type de Document**, souvent abrégé **DTD**, est la première technologie que nous allons étudier pour écrire les définitions de nos documents XML.

Comme nous l'avons déjà précisé dans l'introduction de cette seconde partie, le but est d'écrire une définition de nos documents XML, c'est-à-dire, de construire un ensemble de règles qui vont régir la construction du document XML.

Grâce à l'ensemble de ces règles, nous allons ainsi définir l'architecture de notre document XML et la hiérarchie qui existe entre les balises de celui-ci. Ainsi, on pourra préciser l'enchaînement et le contenu des balises et des attributs contenus dans le document XML.

Finalement, sachez qu'avec les DTD, vous ne pourrez pas toujours tout faire, la technologie commençant en effet à vieillir. Mais comme elle est encore beaucoup utilisée, il est indispensable qu'elle soit étudiée dans ce tutoriel.

Où écrire les DTD ?

Tout comme les fichiers XML, les DTD s'écrivent dans des *fichiers*.

Nous reviendrons sur ce point un peu plus tard, mais sachez dès à présent qu'il existe 2 types de DTD : les **DTD externes** et les **DTD internes**.

Les règles des **DTD internes** s'écrivent *directement dans le fichier XML* qu'elles définissent tandis que les règles des **DTD externes** sont écrites dans un *fichier séparé* portant l'extension **.dtd** .

Maintenant que vous en savez un peu plus, je vous propose de rentrer dans le vif du sujet.

Les éléments

La syntaxe

Pour définir les règles portant sur les **balises**, on utilise le mot clef **ELEMENT**.

```
1 | <!ELEMENT balise (contenu)>
```

Une règle peut donc se découper en 3 mots clefs : ELEMENT, balise et contenu.

Retour sur la balise

Le mot-clef **balise** est à remplacer par le nom de la balise à laquelle vous souhaitez appliquer la règle. Pour exemple, reprenons une balise du TP de la partie 1 :

```
1 | <nom>DÖE</nom>
```

On écrira alors :

```
1 | <!ELEMENT nom (contenu)>
```

Retour sur le contenu

Cet emplacement a pour vocation de décrire ce que doit contenir la balise : est-ce une autre balise ou est-ce une valeur ?

Cas d'une balise en contenant une autre

Par exemple, regardons la règle suivante :

```
1 | <!ELEMENT personne (nom)>
2 | <!-- suite de la DTD -->
```

Cette règle signifie que la balise `<personne />` contient la balise `<nom />`.

Le document XML respectant cette règle ressemble donc à cela :

```
1 | <personne>
2 |   <nom>John DÖE</nom>
3 | </personne>
```



Nous n'avons défini aucune règle pour la balise `<nom/>`. Le document n'est, par conséquent, pas valide. **En effet, dans une DTD, il est impératif de décrire tout le document sans exception.** Des balises qui n'apparaissent pas dans la DTD ne peuvent pas être utilisées dans le document XML.

Cas d'une balise contenant une valeur

Dans le cas où notre balise contient une **valeur simple**, on utilisera la mot clef #PCDATA. Une valeur simple désigne par exemple une chaîne de caractères, un entier, un nombre décimal, un caractère, etc.

En se basant sur l'exemple précédent :

```

1 | <personne>
2 |   <nom>John DOE</nom>
3 | </personne>
```

nous avions déjà défini une règle pour la balise `<personne/>` :

```
1 | <!ELEMENT personne (nom)>
```

Nous pouvons maintenant compléter notre DTD en ajoutant une règle pour la balise `<nom/>`. Par exemple, si l'on souhaite que cette balise contienne une valeur simple, on écrira :

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

Au final, la DTD de notre document XML est donc la suivante :

```

1 | <!ELEMENT personne (nom)>
2 | <!ELEMENT nom (#PCDATA)>
```

Cas d'une balise vide

Il est également possible d'indiquer qu'une balise ne contient rien grâce au mot-clef `EMPTY`. Prenons les règles suivantes :

```

1 | <!ELEMENT personne (nom)>
2 | <!ELEMENT nom EMPTY>
```

Le document XML répondant à la définition DTD précédente est le suivant :

```

1 | <personne>
2 |   <nom />
3 | </personne>
```



À noter : lors de l'utilisation du mot clef `EMPTY`, l'usage des parenthèses n'est pas obligatoire !

Cas d'une balise pouvant tout contenir

Il nous reste un cas à voir : celui d'une balise qui peut tout contenir, c'est à dire, une autre balise, une valeur simple ou tout simplement être vide. Dans ce cas, on utilise le mot-clef `ANY`

Prenons la règle suivante :

```

1 | <!ELEMENT personne (nom)>
2 | <!ELEMENT nom ANY>
```

Les documents XML suivants sont bien valides :

```
1 <!-- valeur simple -->
2 <personne>
3   <nom>John DOE</nom>
4 </personne>
5
6 <!-- vide -->
7 <personne>
8   <nom />
9 </personne>
```



Bien que le mot-clé ANY existe, il est souvent déconseillé de l'utiliser afin de restreindre le plus possible la liberté de rédaction du document XML.



Comme pour le mot-clé EMPTY, l'usage des parenthèses n'est pas obligatoire pour le mot-clé ANY !

Structurer le contenu des balises

Nous allons voir maintenant des **syntaxes** permettant d'apporter un peu de **généricité** aux définitions DTD. Par exemple, un répertoire contient généralement un nombre variable de personnes, il faut donc permettre au document XML d'être valide quel que soit le nombre de personnes qu'il contient.

La séquence

Une **séquence** permet de décrire l'enchaînement imposé des balises. Il suffit d'indiquer le nom des balises en les séparant par des *virgules*.

```
1 | <!ELEMENT balise (balise2, balise3, balise4, balise5, etc.)>
```

Prenons l'exemple suivant :

```
1 | <!ELEMENT personne (nom, prenom, age)>
2 | <!ELEMENT nom (#PCDATA)>
3 | <!ELEMENT prenom (#PCDATA)>
4 | <!ELEMENT age (#PCDATA)>
```

Cette définition impose que la balise `<personne />` contienne obligatoirement les balises `<nom />`, `<prenom />` et `<age />` dans cet ordre. Regardons alors la validité des documents XML qui suivent :

```
1 | <!-- valide -->
2 | <personne>
3 |   <nom>DOE</nom>
```

```
4 |     <prenom>John</prenom>
5 |     <age>24</age>
6 | </personne>
7 |
8 | <!-- invalide -->
9 | <!-- les balises ne sont pas dans le bon ordre -->
10| <personne>
11|     <prenom>John</prenom>
12|     <nom>DOE</nom>
13|     <age>24</age>
14| </personne>
15|
16| <!-- invalide -->
17| <!-- il manque une balise -->
18| <personne>
19|     <prenom>John</prenom>
20|     <age>24</age>
21| </personne>
22|
23| <!-- invalide -->
24| <!-- il y a une balise en trop qui plus est n'est pas déclarée
   -->
25| <personne>
26|     <nom>DOE</nom>
27|     <prenom>John</prenom>
28|     <age>24</age>
29|     <date>12/12/2012</date>
30| </personne>
```

La liste de choix

Une **liste de choix** permet de dire qu'une balise contient l'une des balises décrites. Il suffit d'indiquer le nom des balises en les séparant par une **barre verticale**.

```
1 | <!ELEMENT balise (balise2 | balise3 | balise4 | balise5 | etc.)>
```

Prenons l'exemple suivant :

```
1 | <!ELEMENT personne (nom | prenom)>
2 | <!ELEMENT nom (#PCDATA)>
3 | <!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<personne />` contienne obligatoirement la balise `<nom />` ou la balise `<prenom />`.

Regardons alors la validité des documents XML ci-dessous :

```
1 | <!-- valide -->
2 | <personne>
3 |     <nom>DOE</nom>
```

```
4 | </personne>
5 |
6 | <!-- valide -->
7 | <personne>
8 |     <prenom>John</prenom>
9 | </personne>
10 |
11 | <!-- invalide -->
12 | <!-- les 2 balises prenom et nom ne peuvent pas être présentes
13 |     en même temps. -->
14 | <personne>
15 |     <prenom>John</prenom>
16 |     <nom>DOE</nom>
17 | </personne>
18 | <!-- invalide -->
19 | <!-- il manque une balise -->
20 | <personne />
```

La balise optionnelle

Une balise peut être **optionnelle**. Pour indiquer qu'une balise est optionnelle, on fait suivre son nom par un **point d'interrogation**.

```
1 | <!ELEMENT balise (balise2, balise3?, balise4)>
```

Prenons l'exemple suivant :

```
1 | <!ELEMENT personne (nom, prenom?)>
2 | <!ELEMENT nom (#PCDATA)>
3 | <!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<personne />` contienne obligatoirement la balise `<nom />` puis éventuellement `<prenom />`. Regardons alors la validité de ces documents XML :

```
1 | <!-- valide -->
2 | <personne>
3 |     <nom>DOE</nom>
4 | </personne>
5 |
6 | <!-- valide -->
7 | <personne>
8 |     <nom>DOE</nom>
9 |     <prenom>John</prenom>
10 | </personne>
11 |
12 | <!-- invalide -->
13 | <!-- l'ordre des balises n'est pas respecté -->
14 | <personne>
```

```

15 |     <prenom>John</prenom>
16 |     <nom>DOE</nom>
17 | </personne>
```

La balise répétée optionnelle

Une balise peut être **répétée plusieurs fois** même si elle est optionnelle. Pour indiquer une telle balise, on fait suivre son nom par une **étoile**.

```
1 | <!ELEMENT balise (balise2, balise3*, balise4)>
```

Soit l'ensemble de règles suivant :

```

1 | <!ELEMENT repertoire (personne*)>
2 | <!ELEMENT personne (nom, prenom)>
3 | <!ELEMENT nom (#PCDATA)>
4 | <!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<repertoire />` contienne entre 0 et une infinité de fois la balise `<personne />`. La balise `<personne />`, quant à elle, doit obligatoirement contenir les balises `<nom />` et `<prenom />` dans cet ordre.

Regardons alors la validité des documents XML :

```

1 | <!-- valide -->
2 | <repertoire>
3 |   <personne>
4 |     <nom>DOE</nom>
5 |     <prenom>John</prenom>
6 |   </personne>
7 |   <personne>
8 |     <nom>POPPINS</nom>
9 |     <prenom>Marie</prenom>
10 |   </personne>
11 | </repertoire>
12
13 <!-- valide -->
14 <repertoire>
15   <personne>
16     <nom>DOE</nom>
17     <prenom>John</prenom>
18   </personne>
19 </repertoire>
20
21 <!-- valide -->
22 <repertoire />
23
24 <!-- invalide -->
25 <!-- il manque la balise prenom dans la seconde balise
     personne-->
```

```
26 | <repertoire>
27 |   <personne>
28 |     <nom>DOE</nom>
29 |     <prenom>John</prenom>
30 |   </personne>
31 |   <personne>
32 |     <nom>POPPINS</nom>
33 |   </personne>
34 | </repertoire>
```

La balise répétée

Une balise peut être **répétée plusieurs fois**. Pour indiquer une telle balise, on fait suivre son nom par un **plus**.

```
1 | <!ELEMENT balise (balise2, balise3+, balise4)>
```

Prenons l'exemple suivant :

```
1 | <!ELEMENT repertoire (personne+)>
2 | <!ELEMENT personne (nom, prenom)>
3 | <!ELEMENT nom (#PCDATA)>
4 | <!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<repertoire />` contienne **au minimum une fois** la balise `<personne />`. La balise `<personne />` quant à elle doit obligatoirement contenir les balises `<nom />` et `<prenom />` dans cet ordre.

Regardons alors la validité des documents XML suivants :

```
1 | <!-- valide -->
2 | <repertoire>
3 |   <personne>
4 |     <nom>DOE</nom>
5 |     <prenom>John</prenom>
6 |   </personne>
7 |   <personne>
8 |     <nom>POPPINS</nom>
9 |     <prenom>Marie</prenom>
10 |   </personne>
11 | </repertoire>
12 |
13 | <!-- invalide -->
14 | <repertoire>
15 |   <personne>
16 |     <nom>DOE</nom>
17 |     <prenom>John</prenom>
18 |   </personne>
19 | </repertoire>
20 | 
```

```
21 | <!-- invalide -->
22 | <!-- la balise personne doit être présente au moins une fois-->
23 | <repertoire />
```

En résumé

- Un **document valide** est un document bien formé conforme à une définition.
- Un **document conforme à une définition** est un document qui respecte toutes les règles qui lui sont imposées dans les fameuses définitions.
- Il existe les **DTD internes** et les **DTD externes** .
- Il est possible d'écrire de nombreuses règles grâce aux **DTD**.
- Le mot clef **ELEMENT** permet de d'écrire les règles relatives aux **balises XML**.

DTD : les attributs et les entités

Difficulté : 

Dans le chapitre précédent, nous avons vu comment décrire les **balises** de nos documents XML, mais ce n'est pas suffisant pour pouvoir décrire l'intégralité d'un **document XML**. En effet, rappelez vous qu'une balise peut contenir ce qu'on appelle des **attributs**. Il convient donc de décrire les règles relatives à ces attributs. C'est ce que nous allons voir au cours de ce chapitre.

Ce chapitre sera également l'occasion de découvrir une nouvelle notion dont je n'ai pas encore parlé : les **entités**. Je ne vous en dis pas plus pour le moment, je préfère garder un peu de suspens autour de la définition de cette notion et son utilisation.



Les attributs

Dans le chapitre précédent, nous avons découvert la syntaxe permettant de définir des règles sur les **balises de nos documents XML**. Vous allez voir que le principe est le même pour définir des **règles à nos attributs**.

La syntaxe

Pour indiquer que notre règle porte sur un **attribut**, on utilise le mot clef **ATTLIST**. On utilise alors la syntaxe suivante :

```
1 | <!ATTLIST balise attribut type mode>
```

Une règle peut donc se diviser en 5 mots clefs : **ATTLIST**, **balise**, **attribut**, **type** et **mode**.

Retour sur la balise et l'attribut

Il n'est pas nécessaire de s'attarder trop longtemps sur le sujet, il suffit simplement d'écrire le nom de la balise et de l'attribut concerné par la règle.

Par exemple, reprenons une balise du TP de la partie 1 :

```
1 | <personne sexe="masculin" />
```

On écrira alors :

```
1 | <!ATTLIST personne sexe type mode>
```

Retour sur le type

Cet emplacement a pour vocation de décrire le **type** de l'attribut. Est-ce une valeur bien précise ? Est-ce du texte ? Un identifiant ?

Cas d'un attribut ayant pour type la liste des valeurs possibles

Nous allons étudier ici le cas d'un **attribut ayant pour type une liste de valeurs**. Les différentes valeurs possibles pour l'attribut sont séparées par une **barre verticale**

```
1 | <!ATTLIST balise attribut (valeur 1 | valeur 2 | valeur 3 | etc  
       .) mode>
```

Reprenons une nouvelle fois la balise `<personne />`. Nous avons vu que cette balise possède un attribut **sex**. Nous allons ici imposer la valeur que peut prendre cette attribut : soit **masculin**, soit **feminin**.

Voici ce à quoi la règle portant sur l'attribut dans notre DTD doit ressembler :

```
1 | <!ATTLIST personne sexe (masculin|féminin) mode>
```

Quelques exemple de documents XML possibles :

```
1 | <!-- valide -->
2 | <personne sexe="masculin" />
3 |
4 | <!-- valide -->
5 | <personne sexe="féminin" />
6 |
7 | <!-- invalide -->
8 | <personne sexe="autre" />
```

Cas d'un attribut ayant pour type du texte non « parsé »

Derrière le terme « **texte non** »**parsé** » se cache en fait la possibilité de mettre ce que l'on veut comme valeur : un nombre, une lettre, une chaîne de caractères, etc. Il s'agit de données qui ne seront pas analysées par le « parseur » au moment de la validation et/ou l'exploitation de votre document XML.

Dans le cas où notre attribut contient du **texte non** « **parsé** », on utilise la mot clef CDATA.

```
1 | <!ATTLIST balise attribut CDATA mode>
```

Soit la règle suivante :

```
1 | <!ATTLIST personne sexe CDATA mode>
```

Notre document XML répondant à cette règle peut ressembler à cela :

```
1 | <!-- valide -->
2 | <personne sexe="masculin" />
3 |
4 | <!-- valide -->
5 | <personne sexe="féminin" />
6 |
7 | <!-- valide -->
8 | <personne sexe="autre" />
9 |
10 | <!-- valide -->
11 | <personne sexe="12" />
```

Cas d'un attribut ayant pour type un identifiant unique

Il est tout à fait possible de vouloir qu'une balise possède un attribut permettant de l'identifier de manière unique.

Prenons par exemple l'exemple d'une course à pied. Dans le classement de la course, il y aura un **unique** vainqueur, un **unique** second et un **unique** troisième.

Pour indiquer que la **valeur de l'attribut est unique**, on utilise le mot clef **ID** comme **IDentifiant**.

```
1 | <!ATTLIST balise attribut ID mode>
```

Prenons par exemple la règle suivante :

```
1 | <!ATTLIST personne position ID mode>
```

Voici quelques exemples de documents XML :

```
1 | <!-- valide -->
2 | <personne position="POS-1" />
3 | <personne position="POS-2" />
4 | <personne position="POS-3" />
5 |
6 | <!-- invalide -->
7 | <personne position="POS-1" />
8 | <personne position="POS-1" />
9 | <personne position="POS-2" />
```

Cas d'un attribut ayant pour type une référence à un identifiant unique

Il est tout à fait possible que dans votre document, un de vos attributs fasse référence à un identifiant. Cela permet souvent de ne pas écrire 100 fois les mêmes informations.

Par exemple, votre document XML peut vous servir à représenter des liens de parenté entre des personnes. Grâce aux références, nous n'allons pas devoir imbriquer des balises XML dans tous les sens pour tenter de représenter le père d'une personne ou le fils d'une personne.

Pour faire référence à un identifiant unique, on utilise le mot clef **IDREF**.

Prenons par exemple la règle suivante :

```
1 | <!ATTLIST father id ID mode >
2 | <!ATTLIST child id ID mode
3 |           father IDREF mode
4 | >
```

Cette règle signifie que la balise personne a 2 attributs : **id** qui est l'identifiant unique de la personne et **father** qui fait référence une autre personne.

Illustrons immédiatement avec un exemple XML :

```
1 | <!-- valide -->
2 | <father id="PER-1" />
3 | <child id="PER-2" father="PER-1" />
4 |
5 | <!-- invalide -->
6 | <!-- l'identifiant PER-0 n'apparaît nulle part -->
7 | <father id="PER-1" />
8 | <child id="PER-2" father="PER-0" />
```

Dans cet exemple, la personne **PER-2** a pour père la personne **PER-1**. Ainsi, on matérialise bien le lien entre ces 2 personnes.

Retour sur le mode

Cet emplacement permet de donner une information supplémentaire sur l'attribut comme par exemple une indication sur son obligation ou sa valeur.

Cas d'un attribut obligatoire

Lorsqu'on souhaite qu'un **attribut soit obligatoirement renseigné**, on utilise le mot clef **#REQUIRED**.

Par exemple, si l'on souhaite que le sexe d'une personne soit renseigné, on utilisera la règle suivante :

```
1 | <!ATTLIST personne sexe (masculin|féminin) #REQUIRED>
```

Voici alors quelques exemples de documents XML possibles :

```
1 | <!-- valide -->
2 | <personne sexe="mASCULIN" />
3 |
4 | <!-- valide -->
5 | <personne sexe="fÉMININ" />
6 |
7 | <!-- invalide -->
8 | <personne />
```

Cas d'un attribut optionnel

Si au contraire on souhaite indiquer qu'un **attribut n'est pas obligatoire**, on utilise le mot clef **#IMPLIED**.

Si l'on reprend l'exemple précédent, on peut indiquer qu'il n'est pas obligatoire de renseigner le sexe d'une personne par la règle suivante :

```
1 | <!ATTLIST personne sexe CDATA #IMPLIED>
```

Voici alors quelques exemples de documents XML possibles :

```
1 | <!-- valide -->
2 | <personne sexe="mASCULIN" />
3 |
4 | <!-- valide -->
5 | <personne sexe="fÉMININ" />
6 |
7 | <!-- valide -->
8 | <personne sexe="15" />
```

```
9 | <!-- valide -->
10| <personne />
11|
```

Cas d'une valeur par défaut

Il est également possible d'indiquer une valeur par défaut pour un attribut. Il suffit tout simplement d'écrire cette valeur « en dur » dans la règle.

Par exemple, il est possible d'indiquer qu'une personne dont l'attribut **sexé** n'est pas renseigné est un homme par défaut grâce à la règle suivante :

```
1 | <!ATTLIST personne sexe CDATA "masculin">
```

Voici alors quelques exemple de documents XML possibles :

```
1 | <!-- valide -->
2 | <personne sexe="masculin" />
3 |
4 | <!-- valide -->
5 | <personne sexe="féminin" />
6 |
7 | <!-- valide -->
8 | <!-- l'attribut sexe vaut "masculin" -->
9 | <personne />
```

Cas d'une constante

Enfin, il est possible de rendre **obligatoire un attribut et de fixer sa valeur** grâce au mot clef **#FIXED** suivi de ladite valeur.

Cette situation peut par exemple se rencontrer lorsque l'on souhaite travailler dans une devise bien précise et que l'on souhaite qu'elle apparaisse dans le document.

Par exemple, la règle suivante permet d'indiquer que la devise doit obligatoirement apparaître et a pour seule valeur possible l'euro.

```
1 | <!ATTLIST objet devise CDATA #FIXED "Euro">
```

Voici alors quelques exemple de documents XML possibles :

```
1 | <!-- valide -->
2 | <objet devise="Euro" />
3 |
4 | <!-- invalide -->
5 | <objet devise="Dollar" />
6 |
7 | <!-- invalide -->
8 | <objet />
```

Les entités

Une autre notion assez importante concernant le DTD est la notion d'**entité**.

Définition

Une **entité** peut-être considérée comme un alias permettant de réutiliser des informations au sein du document XML ou de la définition DTD.

Au cours de ce chapitre, nous reviendrons sur les 3 types d'entités qui existent : les **entités générales**, les **entités paramètres** et les **entités externes**.

Les entités générales

Définition

Les **entités générales** sont les entités les plus simples. Elles permettent d'associer un alias à une information afin de l'utiliser dans le document XML.

La syntaxe

Voyons tout de suite la syntaxe d'une entité générale :

```
1 | <!ENTITY nom "valeur">
```

Pour utiliser une entité générale dans notre document XML, il suffit d'utiliser la syntaxe suivante :

```
1 | &nom;
```

Afin d'illustrer un peu plus clairement mes propos, voyons tout de suite un exemple :

```
1 | <!ENTITY samsung "Samsung">
2 | <!ENTITY apple "Apple">
3 |
4 | <telephone>
5 |   <marque>&samsung;</marque>
6 |   <modele>Galaxy S3</modele>
7 | </telephone>
8 | <telephone>
9 |   <marque>&apple;</marque>
10 |   <modele>iPhone 4</modele>
11 | </telephone>
```

Au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives, ce qui donne une fois interprété :

```
1 | <telephone>
2 |   <marque>Samsung</marque>
```

```
3 |     <modele>Galaxy S3</modele>
4 | </telephone>
5 | <telephone>
6 |     <marque>Apple</marque>
7 |     <modele>iPhone 4</modele>
8 | </telephone>
```

Les entités paramètres

Définition

Contrairement aux entités générales qui apparaissent dans les documents XML, les **entités paramètres** n'apparaissent que dans les définitions DTD. Elles permettent d'associer un alias à une partie de la déclaration de la DTD.

La syntaxe

Voyons tout de suite la syntaxe d'une entité paramètre :

```
1 | <!ENTITY % nom "valeur">
```

Pour utiliser une entité paramètre dans notre DTD, il suffit d'utiliser la syntaxe suivante :

```
1 | %nom;
```

Prenons par exemple ce cas où des téléphones ont pour attribut une marque :

```
1 | <telephone marque="Samsung" />
2 | <telephone marque="Apple" />
```

Normalement, pour indiquer que l'attribut `marque` de la balise `<telephone/>` est obligatoire et qu'il doit contenir la valeur Samsung ou Apple, nous devons écrire la règle suivante :

```
1 | <!ATTLIST telephone marque (Samsung|Apple) #REQUIRED>
```

À l'aide d'une entité paramètre, cette même règle s'écrit de la façon suivante :

```
1 | <!ENTITY % listeMarques "marque (Samsung|Apple) #REQUIRED">
2 | <!ATTLIST telephone %listeMarques; >
```

Encore une fois, au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives.

Les entités externes

Définition

Il existe en réalité 2 types d'entités externes : les **analysées** et les **non analysées**. Dans le cadre de ce cours, nous nous limiterons aux **entités externes analysées**.

Les **entités externes analysées** ont sensiblement le même rôle que les entités générales, c'est à dire qu'elles permettent d'associer un alias à une information afin de l'utiliser dans le document XML. Mais, dans le cas des entités externes analysées, les informations sont stockées dans un fichier séparé.

La syntaxe

Voyons tout de suite la syntaxe d'une entité externe :

```
1 | <!ENTITY nom SYSTEM "URI">
```

Pour utiliser une entité externe dans notre XML, il suffit d'utiliser la syntaxe suivante :

```
1 | &nom;
```

Si l'on reprend notre premier exemple, voici ce que cela donne :

```
1 | <!ENTITY samsung SYSTEM "samsung.xml">
2 | <!ENTITY apple SYSTEM "apple.xml">
3 |
4 | <telephone>
5 |   &samsung;
6 |   <modele>Galaxy S3</modele>
7 | </telephone>
8 | <telephone>
9 |   &apple;
10 |  <modele>iPhone 4</modele>
11 | </telephone>
```

Le contenu des fichiers `samsung.xml` et `apple.xml` sera par exemple le suivant :

```
1 | <!-- Contenu du fichier samsung.xml -->
2 | <marque>Samsung</marque>
3 |
4 | <!-- Contenu du fichier apple.xml -->
5 | <marque>Apple</marque>
```

Au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives, ce qui donne une fois interprété :

```
1 | <telephone>
2 |   <marque>Samsung</marque>
3 |   <modele>Galaxy S3</modele>
4 | </telephone>
5 | <telephone>
```

```
6 |     <marque>Apple</marque>
7 |     <modele>iPhone 4</modele>
8 | </telephone>
```

En résumé

- Le mot clef **ATTLIST** permet d'écrire les règles relatives aux attributs d'une balise.
- Les **entités** permettent de jouer les fainéants en réutilisant des éléments qui reviennent souvent dans un document.

DTD : où les écrire ?

Difficulté : 

Au cours des derniers chapitres, nous avons étudié tout ce qu'il faut savoir ou presque sur les **DTD**. Il vous reste cependant encore une chose à apprendre avant que vous puissiez être indépendant et passer à la pratique : **où écrire les DTD** ?

Ce dernier chapitre avant un TP sera également l'occasion de vous révéler qu'il existe en réalité plusieurs sortes de DTD.



Les DTD internes

Comme je vous l'ai déjà précisé dans le premier chapitre de cette seconde partie, on distingue 2 types de DTD : les **internes** et les **externes**.

Commençons par étudier les **DTD internes**.

Définition

Une **DTD interne** est une DTD qui est écrite dans le même fichier que le document XML. Elle est généralement spécifique au document XML dans lequel elle est écrite.

La syntaxe

Une **DTD interne** s'écrit dans ce qu'on appelle le **DOCTYPE**. On le place sous le prologue du document et au dessus du contenu XML.

Voyons plus précisément la syntaxe :

```
1 | <!DOCTYPE racine [ ]>
```

La **DTD interne** est ensuite écrite entre les **[]**. Dans ce **DOCTYPE**, le mot **racine** doit être remplacé par le nom de la balise qui forme la racine du document XML.

Illustrons avec un exemple

Afin que tout cela vous paraisse moins abstrait, je vous propose de voir un exemple.

Prenons l'énoncé suivant :

Une boutique possède plusieurs téléphones. Chaque téléphone est d'une certaine marque et d'un certain modèle représenté par une chaîne de caractère.

Un document XML répondant à cet énoncé peut être le suivant :

```
1 | <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2 |
3 | <boutique>
4 |   <telephone>
5 |     <marque>Samsung</marque>
6 |     <modele>Galaxy S3</modele>
7 |   </telephone>
8 |
9 |   <telephone>
10|     <marque>Apple</marque>
11|     <modele>iPhone 4</modele>
12|   </telephone>
13| 
```

```

14 |     <telephone>
15 |         <marque>Nokia</marque>
16 |         <modele>Lumia 800</modele>
17 |     </telephone>
18 | </boutique>

```

La définition DTD est la suivante :

```

1 | <!ELEMENT boutique (telephone*)>
2 | <!ELEMENT telephone (marque, modele)>
3 | <!ELEMENT marque (#PCDATA)>
4 | <!ELEMENT modele (#PCDATA)>

```

Le document XML complet avec la DTD interne sera par conséquent le suivant :

```

1 | <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2 |
3 | <!DOCTYPE boutique [
4 |     <!ELEMENT boutique (telephone*)>
5 |     <!ELEMENT telephone (marque, modele)>
6 |     <!ELEMENT marque (#PCDATA)>
7 |     <!ELEMENT modele (#PCDATA)>
8 | ]>
9 |
10 | <boutique>
11 |     <telephone>
12 |         <marque>Samsung</marque>
13 |         <modele>Galaxy S3</modele>
14 |     </telephone>
15 |
16 |     <telephone>
17 |         <marque>Apple</marque>
18 |         <modele>iPhone 4</modele>
19 |     </telephone>
20 |
21 |     <telephone>
22 |         <marque>Nokia</marque>
23 |         <modele>Lumia 800</modele>
24 |     </telephone>
25 | </boutique>

```

Maintenant que vous savez ce qu'est une **DTD interne**, passons sans plus attendre à la **DTD externe** !

Les DTD externes

Définition

Une **DTD externe** est une DTD qui est écrite dans un autre document que le document XML. Si elle est écrite dans un autre document, c'est que souvent, elle est

commune à plusieurs documents XML qui l'exploitent.

De manière générale, afin de bien séparer le contenu XML de sa définition DTD, on prendra l'habitude de créer plusieurs fichiers afin de les séparer.

Un fichier contenant uniquement une DTD porte l'extension **.dtd**.

La syntaxe

L'étude de la **syntaxe d'une DTD externe** est l'occasion de vous révéler qu'il existe en réalité 2 types de DTD : les DTD externes **PUBLIC** et les DTD externes **SYSTEM**.

Dans les 2 cas et comme pour une DTD interne, c'est dans le **DOCTYPE** que cela se passe.

Les DTD externes PUBLIC

Les **DTD externes PUBLIC** sont généralement utilisées lorsque la DTD est une norme. C'est par exemple cas dans les documents xHTML 1.0.

La syntaxe est la suivante :

```
1 | <!DOCTYPE racine PUBLIC "identifiant" "url">
```

Si on l'applique à un document xHTML, on obtient alors le **DOCTYPE** suivant :

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

Pour être honnête, nous n'allons jamais utiliser les **DTD externes PUBLIC** dans ce tutoriel, c'est pourquoi je vous propose de passer immédiatement aux **DTD externes SYSTEM**.

Les DTD externes SYSTEM

Une **DTD externe SYSTEM** permet d'indiquer au document XML l'adresse du document DTD. Cette adresse peut-être *relative* ou *absolue*.

Voyons plus précisément la syntaxe :

```
1 | <!DOCTYPE racine SYSTEM "URI">
```

Afin d'illustrer mes propos, je vous propose de reprendre l'exemple de la boutique de téléphone que j'ai utilisé dans la partie sur la DTD interne.

Voici un rappel de l'énoncé :

Une boutique possède plusieurs téléphones. Chaque téléphone est d'une certaine marque et d'un certain modèle, tous les 2 représentés par une chaîne de caractère.

Pour rappel, voici le fichier XML :

```

1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <boutique>
4   <telephone>
5     <marque>Samsung</marque>
6     <modele>Galaxy S3</modele>
7   </telephone>
8
9   <telephone>
10    <marque>Apple</marque>
11    <modele>iPhone 4</modele>
12  </telephone>
13
14  <telephone>
15    <marque>Nokia</marque>
16    <modele>Lumia 800</modele>
17  </telephone>
18 </boutique>
```

Si la DTD ne change pas, elle doit cependant être placée dans un fichier à part, par exemple le fichier **doc1.dtd**. Voici son contenu :

```

1 <!ELEMENT boutique (telephone*)>
2 <!ELEMENT telephone (marque, modele)>
3 <!ELEMENT marque (#PCDATA)>
4 <!ELEMENT modele (#PCDATA)>
```

Le document XML complet avec la DTD externe sera alors le suivant (on part ici du principe que le fichier XML et DTD sont stockés au même endroit) :

```

1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <!DOCTYPE boutique SYSTEM "doc1.dtd">
4
5 <boutique>
6   <telephone>
7     <marque>Samsung</marque>
8     <modele>Galaxy S3</modele>
9   </telephone>
10
11  <telephone>
12    <marque>Apple</marque>
13    <modele>iPhone 4</modele>
14  </telephone>
15
16  <telephone>
17    <marque>Nokia</marque>
18    <modele>Lumia 800</modele>
19  </telephone>
20 </boutique>
```

Retour sur le prologue

Dans la partie précédente de ce tutoriel, voici ce que je vous avais dit à propos du fait que nos documents XML soient autonomes ou non :

La dernière information présente dans le prologue est **standalone="yes"**.

Cette information permet de savoir si votre document XML est autonome ou si un autre document lui est rattaché.

Il est encore un peu tôt pour vous en dire plus. Nous reviendrons sur cette notion dans la partie 2 du tutoriel. Pour le moment, acceptez le fait que nos documents sont tous autonomes.

Il est maintenant temps de lever le mystère !

Dans le cas d'une **DTD externe**, nos documents XML ne sont plus autonomes, en effet, ils font référence à un autre fichier qui fournit la DTD. Afin que le document contenant la DTD soit bien pris en compte, nous devons l'indiquer en passant simplement la valeur de l'attribut **standalone** à « **no** ».

Voici ce que cela donne :

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="no" ?>
2
3 <!DOCTYPE boutique SYSTEM "doc1.dtd">
4
5 <boutique>
6   <telephone>
7     <marque>Samsung</marque>
8     <modele>Galaxy S3</modele>
9   </telephone>
10
11  <telephone>
12    <marque>Apple</marque>
13    <modele>iPhone 4</modele>
14  </telephone>
15
16  <telephone>
17    <marque>Nokia</marque>
18    <modele>Lumia 800</modele>
19  </telephone>
20 </boutique>
```

Un exemple avec EditiX

Pour clore ce chapitre, je vous propose de voir ensemble comment écrire une **DTD externe SYSTEM avec EditiX**. Pour faire simple, je vous propose de garder l'exemple précédent de la boutique de téléphones.

Création du document XML

La **création du document XML** n'a rien de bien compliqué puisque nous l'avons déjà vu ensemble dans la partie précédente. Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil à la page 21.

Voici le document que vous devez écrire :

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="no" ?>
2
3 <!DOCTYPE boutique SYSTEM "boutique.dtd">
4
5 <boutique>
6   <telephone>
7     <marque>Samsung</marque>
8     <modele>Galaxy S3</modele>
9   </telephone>
10
11  <telephone>
12    <marque>Apple</marque>
13    <modele>iPhone 4</modele>
14  </telephone>
15
16  <telephone>
17    <marque>Nokia</marque>
18    <modele>Lumia 800</modele>
19  </telephone>
20 </boutique>
```

Si vous essayez de lancer la vérification du document, vous devriez normalement obtenir un message d'erreur, comme celui indiqué à la figure 8.1.



FIGURE 8.1 – Message d'erreur indiquant que le document DTD est introuvable

Ce message est, pour le moment, complètement normal puisque nous n'avons pas encore créé notre document DTD.

Création du document DTD

Pour créer un nouveau document, vous pouvez cliquer sur l'icône visible à la figure 8.2, sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier **[Ctrl] + [N]**.

Dans la liste qui s'affiche, sélectionnez DTD (voir la figure 8.3).

Votre document DTD n'est normalement pas vierge. Voici ce que vous devriez avoir :



FIGURE 8.2 – Nouveau document

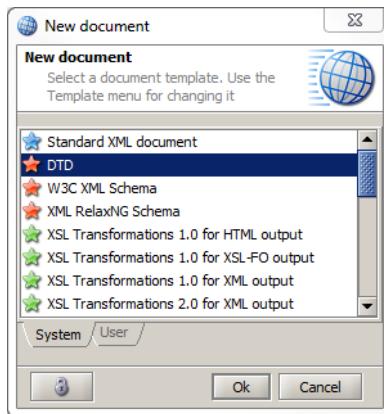


FIGURE 8.3 – Crédation d'un document DTD

```

1 | <!-- DTD created at Wed Sep 12 14:49:47 CEST 2012 with EditiX.
   | Please insert an encoding attribute header for converting
   | any DTD -->
2 |
3 | <!ELEMENT tag (#PCDATA)>
4 | <!ATTLIST tag attribute CDATA #REQUIRED>
```

Replacez le contenu par notre véritable DTD :

```

1 | <!ELEMENT boutique (telephone*)>
2 | <!ELEMENT telephone (marque, modele)>
3 | <!ELEMENT marque (#PCDATA)>
4 | <!ELEMENT modele (#PCDATA)>
```

Enregistrez ensuite votre document avec le nom **boutique.dtd** au même endroit que votre document XML.

Vérification de la DTD

Vous pouvez vérifier que votre DTD n'a pas d'erreur de syntaxe en cliquant sur l'icône visible à la figure 8.4 ou bien en sélectionnant dans la barre de menu DTD/Schema puis Check this DTD ou encore en utilisant le raccourci clavier **Ctrl** + **K**.



FIGURE 8.4 – Vérification de la syntaxe

Vous devriez normalement avoir un message d'information (voir figure 8.5).

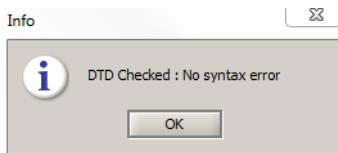


FIGURE 8.5 – Message indiquant que le document DTD est correcte

Vérification du document XML

Il est maintenant temps de vérifier que le document XML est **valide**!

Pour ce faire, sélectionnez dans la barre de menu XML puis Check this document ou encore en utilisant le raccourci clavier [Ctrl] + [K].

Le message visible à la figure 26.4 doit normalement s'afficher.

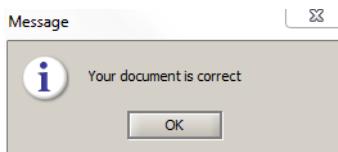


FIGURE 8.6 – Message indiquant que le document XML est valide

En résumé

- Les **DTD internes** s'écrivent dans le document XML.
- Les **DTD externes** s'écrivent dans un fichier différent de celui du document XML dont l'extension est **.dtd**.
- **EditiX** permet en quelques clics de vérifier qu'un document DTD est correct et qu'un document XML est valide.

TP : définition DTD d'un répertoire

Difficulté : 

Votre apprentissage des DTD arrive donc à son terme et rien ne vaut un TP pour le conclure ! Je vous propose donc de réaliser la définition DTD d'un répertoire. L'objectif est de mettre en pratique toutes les notions vues dans les parties précédentes sur les DTD.



L'énoncé

Le but de ce TP est de créer la DTD du répertoire élaboré dans le premier TP.

Pour rappel, voici les informations que l'on souhaite connaître pour chaque personne :

- Son sexe (homme ou femme).
- Son nom.
- Son prénom.
- Son adresse.
- Un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.).
- Une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici le document XML que nous avions construit :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <repertoire>
4     <!-- John DOE -->
5     <personne sexe="masculin">
6         <nom>DOE</nom>
7         <prenom>John</prenom>
8         <adresse>
9             <numero>7</numero>
10            <voie type="impasse">impasse du chemin</voie>
11            <codePostal>75015</codePostal>
12            <ville>PARIS</ville>
13            <pays>FRANCE</pays>
14        </adresse>
15        <telephones>
16            <telephone type="fixe">01 02 03 04 05</telephone>
17            <telephone type="portable">06 07 08 09 10</
18                telephone>
19        </telephones>
20        <emails>
21            <email type="personnel">john.doe@wanadoo.fr</email>
22            <email type="professionnel">john.doe@societe.com</
23                email>
24        </emails>
25    </personne>
26
27    <!-- Marie POPPINS -->
28    <personne sexe="feminin">
29        <nom>POPPINS</nom>
30        <prenom>Marie</prenom>
31        <adresse>
32            <numero>28</numero>
33            <voie type="avenue">avenue de la république</voie>
34            <codePostal>13005</codePostal>
35            <ville>MARSEILLE</ville>
36            <pays>FRANCE</pays>
```

```
35      </adresse>
36      <telephones>
37          <telephone type="bureau">04 05 06 07 08</telephone>
38      </telephones>
39      <emails>
40          <email type="professionnel">contact@poppins.fr</
41              email>
42      </emails>
43  </personne>
</repertoire>
```

Une dernière consigne : la DTD doit être une DTD externe !

Une solution

Une fois de plus, je vous fais part de ma solution !

Le fichier XML avec le DOCTYPE :

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2
3  <!DOCTYPE repertoire SYSTEM "repertoire.dtd">
4
5  <repertoire>
6      <!-- John DOE -->
7      <personne sexe="masculin">
8          <nom>DOE</nom>
9          <prenom>John</prenom>
10         <adresse>
11             <numero>7</numero>
12             <voie type="impasse">impasse du chemin</voie>
13             <codePostal>75015</codePostal>
14             <ville>PARIS</ville>
15             <pays>FRANCE</pays>
16         </adresse>
17         <telephones>
18             <telephone type="fixe">01 02 03 04 05</telephone>
19             <telephone type="portable">06 07 08 09 10</
20                 telephone>
21         </telephones>
22         <emails>
23             <email type="personnel">john.doe@wanadoo.fr</email>
24             <email type="professionnel">john.doe@societe.com</
25                 email>
26         </emails>
27     </personne>
28
29     <!-- Marie POPPINS -->
<personne sexe="feminin">
    <nom>POPPINS</nom>
```

```
30      <prenom>Marie</prenom>
31      <adresse>
32          <numero>28</numero>
33          <voie type="avenue">avenue de la république</voie>
34          <codePostal>13005</codePostal>
35          <ville>MARSEILLE</ville>
36          <pays>FRANCE</pays>
37      </adresse>
38      <telephones>
39          <telephone type="professionnel">04 05 06 07 08</
40              telephone>
41      </telephones>
42      <emails>
43          <email type="professionnel">contact@poppins.fr</
44              email>
45      </emails>
46  </personne>
47 </repertoire>
```

Le fichier DTD :

```
1  <!-- Racine -->
2  <!ELEMENT repertoire (personne*)>
3
4  <!-- Personne -->
5  <!ELEMENT personne (nom, prenom, adresse, telephones, emails)>
6  <!ATTLIST personne sexe (masculin | feminin) #REQUIRED>
7
8  <!-- Nom et prénom -->
9  <!ELEMENT nom (#PCDATA)>
10 <!ELEMENT prenom (#PCDATA)>
11
12 <!-- Bloc adresse -->
13 <!ELEMENT adresse (numero, voie, codePostal, ville, pays)>
14 <!ELEMENT numero (#PCDATA)>
15
16 <!ELEMENT voie (#PCDATA)>
17 <!ATTLIST voie type CDATA #REQUIRED>
18
19 <!ELEMENT codePostal (#PCDATA)>
20 <!ELEMENT ville (#PCDATA)>
21 <!ELEMENT pays (#PCDATA)>
22
23 <!-- Bloc téléphone -->
24 <!ELEMENT telephones (telephone+)>
25 <!ELEMENT telephone (#PCDATA)>
26 <!ATTLIST telephone type CDATA #REQUIRED>
27
28 <!-- Bloc email -->
29 <!ELEMENT emails (email+)>
30 <!ELEMENT email (#PCDATA)>
```

31 | <!ATTLIST email type CDATA #REQUIRED>

▷ Copier la correction
Code web : [786477](#)

Un bref commentaire

Dans cette solution, je suis allé au plus simple en indiquant que pour les types de téléphones, d'e-mails et de voies, j'accepte toutes les chaînes de caractères. Libre à vous de créer de nouvelles règles si vous souhaitez que, par exemple, le choix du type de la voie ne soit possible qu'entre rue, avenue, impasse, etc.

Chapitre 10

Schéma XML : introduction

Difficulté : 

Dans les chapitres précédents, nous avons étudié l'une des technologies permettant d'écrire les définitions de documents XML : les **DTD**. Mais, comme je vous le disais dans le tout premier chapitre, une autre technologie permet elle aussi d'écrire des définitions : les **Schémas XML**.

Cette seconde technologie offre davantage de possibilités que les DTD, il va donc falloir vous accrocher !



I était un fois...

Les défauts des DTD

Peut-être l'avez vous remarqué dans les précédents chapitres, mais les DTD ont quelques défauts.

Un nouveau format

Tout d'abord, les DTD ne sont pas au format XML. Nous avons dû apprendre un nouveau langage avec sa propre syntaxe et ses propres règles.

La principale conséquence est que, pour exploiter une DTD, nous allons être obligé d'utiliser un outil différent de celui qui exploite un fichier XML. Il est vrai que dans notre cas, nous avons utilisé le même outil, à savoir **EditiX**, mais vos futurs programmes, logiciels ou applications mobiles devront forcément exploiter la DTD et le fichier XML différemment, à l'aide, par exemple, d'une API différente.

Le typage de données

Le second défaut que l'on retiendra dans ce cours est que les DTD ne permettent pas de typer des données. Comme vous avez pu le voir, on se contente d'indiquer qu'une balise contient des données, mais impossible de préciser si l'on souhaite que ça soit un nombre entier, un nombre décimal, une date, une chaîne de caractères, etc.

Les apports des schémas XML

C'est pour pallier les défauts des DTD que les **Schémas XML** ont été créés. S'ils proposent au minimum les mêmes fonctionnalités que les DTD, ils en apportent également de nouvelles. En voici quelques unes pêle-mêle.

Le typage des données

Les **Schémas XML** permettent tout d'abord de *typer* les données. Nous verrons également dans la suite de ce tutoriel, qu'il est possible d'aller plus loin en créant nos propres types de données.

Les contraintes

Nous découvrirons aussi que les **Schémas XML** permettent d'être beaucoup plus précis que les DTD lors de l'écriture des différentes contraintes qui régissent un document XML.

Des définitions XML

Un des principaux avantages des **Schémas XML** est qu'ils s'écrivent grâce au XML. Ainsi, pour exploiter un document XML et le Schéma qui lui est associé, vous n'avez en théorie plus besoin de plusieurs outils. Dorénavant un seul suffit !

Structure d'un schéma XML

Maintenant que vous en savez un peu plus sur les **Schémas XML**, je vous propose de voir les bases qui permettent de définir un Schéma XML.

L'extension du fichier

Comme pour les DTD, nous prendrons l'habitude de séparer les données formatées avec XML et le Schéma XML associé dans 2 fichiers distincts.

Bien que c'est les Schémas XML soient écrits avec un langage de type XML, le fichier n'a pas cette extension. Un fichier dans lequel est écrit un Schéma XML porte l'extension « **.xsd** ».

Le prologue

Puisque c'est le XML qui est utilisé, il ne faut pas déroger à la règle du **prologue**.

Ainsi, la première ligne d'un Schéma XML est :

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

Je ne détaille pas ici les différents éléments du prologue puisque je l'ai déjà fait lors de la première partie dans le chapitre traitant de la structure d'un document XML. Si vous avez des doutes, je vous encourage vivement à relire cette partie !

Le corps

Comme pour un fichier XML classique, le **corps d'un Schéma XML** est constitué d'un ensemble de **balises** dont nous verrons le rôle dans les prochains chapitres.

Cependant, une chose ne change pas : la présence d'un **élément racine**, c'est-à-dire la présence d'une balise qui contient toutes les autres. Mais, contrairement à un fichier XML, son nom nous est imposé.

```
1 | <!-- Prologue -->
2 | <?xml version="1.0" encoding="UTF-8" ?>
3 |
4 | <!-- Élément racine -->
5 | <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
6 |
```

```
7 | </xsd:schema>
```

Comme vous pouvez le voir dans le code précédent, l'élément racine est `<xsd:schema />`.

Si l'on regarde de plus près, on remarque la présence de l'attribut `xmlns:xsd`. `xmlns` nous permet de déclarer un **espace de noms**. Si ce vocabulaire ne vous parle pas, je vous encourage à lire le chapitre dédié à cette notion en annexe de ce tutoriel.

A travers la déclaration de cet **espace de noms**, tous les éléments doivent commencer par `xsd:`.

Référencer un schéma XML

Le **référencement d'un schéma XML** se fait au niveau de l'élément racine du fichier XML grâce à l'utilisation de 2 attributs.

L'espace de noms

```
1 | xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

La location

Le second attribut nous permet d'indiquer à notre fichier XML où se situe le fichier contenant le Schéma XML.

2 possibilités s'offrent alors à nous : les schémas XML qui décrivent un espace de noms et ceux qui ne décrivent pas un espace de noms.

Schéma XML décrivant un espace de noms

```
1 | xsi:schemaLocation="chemin_vers_fichier.xsd">
```

Schéma XML ne décrivant pas un espace de noms

Dans les prochains chapitre, c'est ce type de Schéma XML que nous allons utiliser.

On utilisera alors la syntaxe suivante :

```
1 | xsi:noNamespaceSchemaLocation="chemin_vers_fichier.xsd">
```

Pour résumer

Pour résumer, voici ce à quoi nos fichiers XML ressembleront :

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 |
```

```
3 | <racine xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 |     xsi:noNamespaceSchemaLocation="chemin_vers_fichier.xsd"
5 |
6 | </racine>
```

En résumé

- Les **Schémas XML** offrent plus de possibilités que les DTD.
- Les **Schémas XML** s'écrivent à l'aide d'un langage de type XML.
- Un fichier dans lequel est écrit un **Schéma XML** porte l'extension « **.xsd** ».

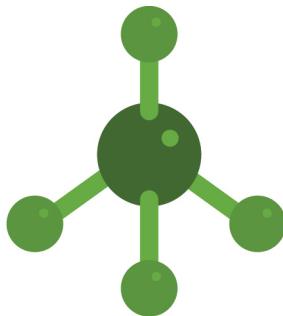
Chapitre 11

Schéma XML : les éléments simples

Difficulté : 

Dans le chapitre précédent, nous avons vu que les **Schémas XML** permettent de pallier les faiblesses des **DTD**. Après avoir également vu la structure d'un Schéma XML, je vous propose d'aborder l'**écriture d'un Schéma XML** à proprement parler.

La première notion que nous allons aborder ce que l'on appelle les **éléments simples**.



Les éléments simples

Définition

Un **élément simple** est un élément qui ne contient qu'une valeur dont le type est dit **simple**. Il ne contient pas d'autres éléments.

Un **élément simple** peut donc être une balise qui ne contient aucun attribut et dans laquelle aucune autre balise n'est imbriquée. Un attribut d'une balise peut également être considéré comme un élément simple. En effet, la valeur d'un attribut est un type simple.

Nous verrons la liste complète des types simples un peu plus loin dans ce tutoriel, mais je peux déjà vous citer quelques exemple afin de tenter d'éclaircir les choses. Un **type simple**, c'est par exemple un chiffre, une date ou encore une chaîne de caractères.

Quelques exemples

Prenons quelques exemples de fichiers XML, et regardons ensemble s'ils peuvent être considérés comme des types simples :

```
1 <!-- Ne contient ni attribut ni aucun autre élément => élément
2   simple -->
3   <nom>ROBERT</nom>
4
5 <!-- Contient un attribut => n'est pas un élément simple -->
6 <!-- Cependant l'attribut "sexe" est un élément simple -->
7 <personne sexe="masculin">Robert DUPONT</personne>
8
9 <!-- La balise personne contient d'autres éléments (les balises
10   nom et prénom) => n'est pas un élément simple -->
11 <personne>
12   <!-- Ne contient ni attribut ni aucun autre élément => élé-
13     ment simple -->
14   <nom>DUPONT</nom>
15
16   <!-- Ne contient ni attribut ni aucun autre élément => élé-
17     ment simple -->
18   <prenom>Robert</prenom>
19 </personne>
```

J'espère que ces exemples vous permettent de mieux comprendre ce qu'est un élément simple.

Déclarer une balise comme un élément simple

Si vous souhaitez déclarer une balise en tant qu'**élément simple**, c'est le mot clef **element** qu'il faut utiliser. N'oubliez pas de précéder son utilisation par **xsd** :

Cette balise prend 2 attributs : un nom et un type.

```
1 | <xsd:element name="mon_nom" type="xsd:mon_type" />
```

Voyons tout de suite un exemple. Soit les éléments simples suivants :

```
1 | <nom>DUPONT</nom>
2 | <prenom>Robert</prenom>
3 | <age>38</age>
```

Au sein d'un Schéma XML, les éléments précédents seront déclarés de la sorte :

```
1 | <xsd:element name="nom" type="xsd:string" />
2 | <xsd:element name="prenom" type="xsd:string" />
3 | <xsd:element name="age" type="xsd:int" />
```



Que sont ces int et ces strings ?

String est utilisé pour qualifier **une chaîne de caractères** et **int** est utilisé pour qualifier **un nombre entier**.

La liste complète des types qu'il est possible d'utiliser est fournie un peu plus loin dans ce tutoriel.

Valeur par défaut et valeur inchangeable

Avant que l'on regarde ensemble la liste des types, j'aimerais revenir sur 2 concepts qui sont les **valeurs par défaut** et les **valeurs inchangeables**.

Valeur par défaut

Comme c'était déjà le cas dans les DTD, il est tout à fait possible d'indiquer dans les Schémas XML qu'un élément a une valeur par défaut. Pour rappel, la **valeur par défaut** est la valeur que va prendre automatiquement un élément si aucune valeur n'est indiquée au niveau du fichier XML.

Pour indiquer une valeur par défaut, c'est l'attribut **default** qui est utilisé au niveau de la balise `<element />` du Schéma XML. Par exemple, si je souhaite indiquer qu'à défaut d'être renseigné, le prénom d'une personne est Robert, je vais écrire la règle suivante :

```
1 | <xsd:element name="prenom" type="xsd:string" default="Robert" /
   >
```

Voici alors quelques exemples de documents XML possibles :

```
1 | <!-- valide -->
2 | <prenom>Jean</prenom>
```

```
3 | <!-- valide -->
4 | <prenom>Marie</prenom>
5 |
6 | <!-- valide -->
7 | <!-- la balise prenom vaut "Robert" -->
8 | <prenom />
```

Valeur constante

S'il est possible d'indiquer une valeur par défaut, il est également possible d'imposer une valeur. Cette valeur inchangable est appelée **constante**.

Pour indiquer une **valeur constante**, c'est l'attribut **fixed** qui est utilisé au niveau de la balise `<element />` du Schéma XML. Par exemple, si je souhaite obliger toute les personnes de mon document XML à porter le prénom Robert, voici la règle à écrire :

```
1 | <xsd:element name="prenom" type="xsd:string" fixed="Robert" />
```

Voyons alors la validité des lignes XML suivantes :

```
1 | <!-- valide -->
2 | <prenom>Robert</prenom>
3 |
4 | <!-- invalide -->
5 | <prenom>Marie</prenom>
6 |
7 | <!-- invalide -->
8 | <prenom/>
```

Les attributs

Comme je vous le disais un peu plus tôt dans ce tutoriel, dans un Schéma XML, tous les **attributs d'une balise XML** sont considérés comme des éléments simples. En effet, ils ne peuvent prendre comme valeur qu'un type simple, c'est-à-dire un nombre, une chaîne de caractère, une date, etc.

Déclarer un attribut

Bien qu'un attribut soit un **élément simple**, nous n'allons pas utiliser le mot clef **element** pour déclarer un attribut. C'est le mot **attribut** qui est utilisé. Encore une fois, n'oubliez pas de faire précéder son utilisation par **xsd** :

Cette balise prend 2 attributs : un *nom* et un *type*.

```
1 | <xsd:attribut name="mon_nom" type="xsd:mon_type" />
```

Prenons par exemple la ligne XML suivante :

```
1 | <personne sexe="masculin">Robert DUPONT</personne>
```

Je ne vais pas détailler ici comment déclarer la balise. En effet, puisqu'elle contient un attribut, c'est ce qu'on appelle un **élément complexe** et nous verrons comment faire un peu plus tard. Cependant, voici comment déclarer l'attribut dans notre Schéma XML :

```
1 | <xsd:attribut name="sexe" type="xsd:string" />
```

Valeur par défaut, obligatoire et inchangeable

Valeur par défaut

Comme c'était déjà le cas dans les DTD, il est tout à fait possible d'indiquer dans les Schémas XML qu'un **attribut a une valeur par défaut**. Pour rappel, la valeur par défaut est la valeur prise automatiquement par un attribut si aucune valeur n'est indiquée au niveau du fichier XML.

Pour indiquer une **valeur par défaut**, c'est l'attribut **default** qui est utilisé au niveau de la balise `<attribut />` du Schéma XML. Par exemple, si je souhaite indiquer qu'à défaut d'être renseigné, le prénom d'une personne est Robert, je vais écrire la règle suivante :

```
1 | <xsd:attribut name="prenom" type="xsd:string" default="Robert" />
```

Valeur constante

S'il est possible d'indiquer une valeur par défaut, il est également possible d'imposer une valeur. Cette valeur inchangeable est appelée **constante**.

Pour indiquer une **valeur constante**, c'est l'attribut **fixed** qui est utilisé au niveau de la balise `<attribut />` du Schéma XML. Par exemple, si je souhaite obliger toutes les personnes de mon document XML à porter le prénom Robert, voici la règle à écrire :

```
1 | <xsd:attribut name="prenom" type="xsd:string" fixed="Robert" />
```

Attribut obligatoire

Tels que nous les déclarons depuis le début de ce chapitre, les attributs sont, par défaut, optionnels.

Pour indiquer qu'un attribut est *obligatoire*, nous devons renseigner la propriété **use** à laquelle nous affectons la valeur **required**. Par exemple, si je souhaite obliger l'utilisation de l'attribut **prenom**, voici la règle à écrire :

```
1 | <xsd:attribut name="prenom" type="xsd:string" use="required" />
```

En résumé

- Un **élément simple** est un élément qui ne contient qu'une valeur dont le type est dit simple comme par exemple une balise qui ne contient aucun attribut et dans laquelle aucune autre balise n'est imbriquée.
- Pour décrire un élément simple, on utilise la balise `<xsd:element />`.
- Il est possible de définir une valeur par défaut ou une valeur constante à un élément simple.
- Un **attribut** est également un élément simple.
- Pour décrire un **attribut**, on utilise la balise `<xsd:attribut />`.
- Il est possible de définir une valeur par défaut, une valeur constante à un élément simple ou rendre obligatoire un attribut.

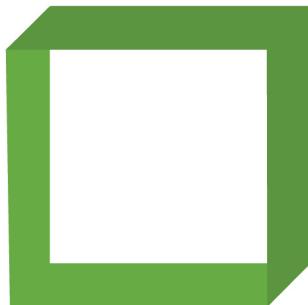
Chapitre 12

Schéma XML : les types simples

Difficulté : 

Dans le chapitre précédent, nous avons vu comment décrire les **éléments simples** et les **attributs**. Nous avons également appris que les valeurs possibles sont des valeurs dites de **types simples** comme par exemple un nombre entier ou une chaîne de caractères.

Dans ce chapitre, nous allons voir en détails les 4 grandes catégories des types simples, à savoir les les **chaînes de caractères**, les **dates** et les **nombres**. Ils manque une catégorie, me direz-vous. Vous avez raison ! Une quatrième catégorie intitulée « divers », regroupe les autres types simples qu'il est possible d'utiliser.



Les types chaînes de caractères

Le tableau récapitulatif

Type	Description	Commentaire
string	représente une chaîne de caractères	attention aux caractères spéciaux
normalizedString	représente une chaîne de caractères normalisée	basé sur le type string
token	représente une chaîne de caractères normalisée sans espace au début et à la fin	basé sur le type normalizedString
language	représente le code d'une langue	basé sur le type token
NMTOKEN	représente une chaîne de caractère "simple"	basé sur le type token applicable uniquement aux attributs
NMTOKENS	représente une liste de NMTOKEN	applicable uniquement aux attributs
Name	représente un nom XML	basé sur le type token
NCName	représente un nom XML sans le caractère :	basé sur le type Name
ID	représente un identifiant unique	basé sur le type NCName applicable uniquement aux attributs
IDREF	référence à un identifiant	basé sur le type NCName applicable uniquement aux attributs
IDREFS	référence une liste d'identifiants	applicable uniquement aux attributs
ENTITY	représente une entité d'un document DTD	basé sur le type NCName applicable uniquement aux attributs
ENTITIES	représente une liste d'entités	applicable uniquement aux attributs

Plus en détails

Le type string

Le type **string** est l'un des premiers types que nous ayons vu ensemble. Il représente une chaîne de caractères et peut donc contenir un peu tout et n'importe quoi. Il est cependant important de noter que certains caractères spéciaux comme le & doivent être écrits avec leur notation HTML.

Une liste des caractères spéciaux et de leur notation HTML est disponible sur le site de CommentÇaMarche.

▷ [Voir la liste](#)
Code web : [888700](#)

Bien que nous ayons déjà vu plusieurs exemples ensemble, je vous en propose un nouveau afin que ce type n'ait plus aucun secret pour vous. Soit la règle de Schéma XML suivante :

```
1 | <xsd:element name="string" type="xsd:string" />
```

Les différentes lignes XML ci-dessous sont alors valides :

```
1 | <string>France</string>
2 |
3 | <string>Site du zéro !</string>
4 |
5 | <string>&</string>
```

Le type **normalizedString**

Le type **normalizedString** est basé sur le type **string** et représente une chaîne de caractères *normalisée*, c'est-à-dire, une chaîne de caractères qui peut contenir tout et n'importe quoi à l'exception de tabulations, de sauts de ligne et de retours chariot. Dans la pratique, il n'est pas interdit de les écrire, mais ils seront automatiquement remplacés par des espaces.

Puisque le type **normalizedString** est basé sur le type **string**, toutes les règles du type **string** s'applique également au type **normalizedString**. Ainsi, les caractères spéciaux comme le **&** doivent être écrits avec leur notation HTML.

Je ne le préciserai pas à chaque fois, mais cette règle est toujours vraie. *Un type hérite toujours de toutes les règles du type sur lequel il se base.*

Le type **token**

Le type **token** est basé sur le type **normalizedString** et représente une chaîne de caractères normalisée sans espace au début ni à la fin. Une nouvelle fois, dans la pratique, il n'est pas interdit de les écrire. Les espaces présents au début et à la fin seront automatiquement supprimés.

Le type **language**

Le type **language** est basé sur le type **token** et représente, comme son nom le laisse deviner, une langue. Cette langue doit être identifiée par 2 lettres (selon la norme ISO 639). Ces 2 caractères peuvent éventuellement être suivi d'un code pays (selon la norme ISO 3166).

Considérons la règle suivante :

```
1 | <xsd:element name="langue" type="xsd:language" />
```

Les différentes lignes XML ci-dessous sont alors valides :

```
1 | <langue>fr</langue>
2 |
3 | <langue>en</langue>
4 |
5 | <langue>en-GB</langue>
6 |
7 | <langue>en-US</langue>
```

Le type NMTOKEN

Le type **NMTOKEN** est basé sur le type **token** et représente une chaîne de caractères simple, c'est-à-dire une chaîne de caractères sans espace qui ne contient que les symboles suivants :

- Des lettres.
- Des chiffres.
- Les caractères spéciaux . - _ et :

Si la chaîne de caractères contient des espaces au début ou à la fin, ils seront automatiquement supprimés.

Afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **NMTOKEN** que pour un attribut.

Le type NMOKENS

Le type **NMOKENS** représente une liste de **NMTOKEN** séparés par un espace. Une nouvelle fois, afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **NMOKENS** seulement pour un attribut.

Soit la règle suivante issue d'un Schéma XML :

```
1 | <xsd:attribut name="liste" type="xsd:NMOKENS" />
```

Les différentes lignes XML ci-dessous sont alors valides :

```
1 | <balise list="A:1_B C-2.">contenu de la balise</balise>
2 |
3 | <balise list="AZERTY 123456 QSDFGH">contenu de la balise</
   balise>
```

Le type Name

Le type **Name** est basé sur le type **token** et représente un **nom XML**, c'est-à-dire une chaîne de caractères sans espace qui ne contient que les symboles suivants :

- Des lettres.
- Des chiffres.
- Les caractères spéciaux . - _ et :

La différence avec le type **NMTOKEN** est qu'une chaîne de caractères de type **Name** doit obligatoirement commencer par une lettre, ou l'un des 2 caractères spéciaux suivants : _ et :

Le type NCName

Le type **NCName** est basé sur le type **Name**. Il hérite donc de toutes les règles du type **Name** auxquelles une nouvelle règle doit être ajoutée : le type **NCName** ne peut pas contenir le caractère spécial :

Le type ID

Le type **ID** est basé sur le type **NCName**, il hérite donc de toutes les règles de ce type. Comme son nom le laisse deviner, un **ID** représente un identifiant. Il doit donc contenir des valeurs uniques. A ce titre, il est impossible de lui définir une valeur fixe ou par défaut.

Comme pour d'autres types vu précédemment, un **ID** ne doit être utilisé qu'avec les attributs afin d'assurer une compatibilité entre les Schémas XML et les DTD.

Le type IDREF

Le type **IDREF** fait référence à un **ID** existant dans le document XML. Tout comme le type **ID**, il est basé sur le type **NCName** et hérite donc de toutes les règles de ce type. Puisque le type **ID** n'est utilisable qu'avec des attributs, il en est naturellement de même pour le type **IDREF**.

Le type IDREFS

Si le type **NMTOKENS** représente une liste de **NMTOKEN** séparés par un espace, le type **IDREFS** représente lui une liste de **IDREF** séparés par un espace.

Afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **IDREFS** que pour un attribut.

Illustrons son utilisation avec un exemple :

```
1 | <xsd:attribut name="enfants" type="xsd:IDREFS" />
2 |
3 | <personne num="P1">Paul</personne>
4 | <personne num="P2">Marie</personne>
5 |
6 | <personne enfants="P1 P2">Jeanne</personne>
```

Le type ENTITY

Le type **ENTITY** permet de faire référence à une entité le plus souvent non XML et déclaré dans des fichiers DTD. Ce type est basé sur le type **NCName**, il hérite donc de toutes ses règles.

Une nouvelle fois, afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **ENTITY** que pour un attribut.

Une nouvelle fois, je vous propose d'illustrer son utilisation par un exemple :

```
1 | <xsd:attribut name="marque" type="xsd:ENTITY" />
```

```
1 | <!ENTITY samsung "Samsung">
2 | <!ENTITY apple "Apple">
3 |
4 | <telephone marque="apple">iPhone</personne>
5 | <telephone marque="samsung">Galaxy SII</personne>
```

Le type ENTITIES

Finalement, le dernier type que nous verrons dans cette catégorie est le type **ENTITIES**. Il permet de faire référence à une liste d'**ENTITY** séparés par un espace.

Puisque c'était déjà le cas pour le type **ENTITY**, le type **ENTITIES** n'échappe pas à la règle et ne doit être utilisé qu'avec un attribut.

Les types dates

Le tableau récapitulatif

Type	Description
duration	représente une durée
date	représente une date
time	représente une heure
dateTime	représente une date et un temps
gYear	représente une année
gYearMonth	représente une année et un mois
gMonth	représente un mois
gMonthDay	représente un mois et un jour
gDay	représente un jour

Plus en détails

Le type duration

Le type **duration**, comme son nom le laisse deviner, représente une *durée*. Cette durée s'exprime en nombre d'années, de mois, de jours, d'heures, de minutes et de secondes selon une expression qui n'est pas des plus simples à savoir PnYnMnDTnHnMnS.

Je vous propose de la décortiquer :

- P marque le début de l'expression.
- nY représente le nombre d'années (year) où n est un nombre entier.
- nM représente le nombre de mois (month) où n est un nombre entier.
- nD représente le nombre de jours (day) où n est un nombre entier.
- T permet de séparer la partie date de l'expression de sa partie heure.
- nH représente le nombre d'heures (hour) où n est un nombre entier.
- nM représente le nombre de minutes (minute) où n est un nombre entier.
- nS représente le nombre de secondes (second) où n est un nombre entier ou décimal.

L'expression peut-être précédé du signe - dans le cas où l'on souhaite exprimer une durée négative. Bien évidemment, tous les champs ne doivent pas forcément être renseignés. Ainsi, il est possible de ne renseigner que les heures, les minutes, etc. Dans le cas où l'expression n'exprime qu'une date, le symbole T ne doit plus figurer.

Je vous accorde que toutes ces règles ne sont pas facile à assimiler, c'est pourquoi je vous propose de voir quelques exemples :

```
1 | <xsd:element name="duree" type="xsd:duration" />
2 |
3 | <!-- 42 ans et 6 minutes -->
4 | <duree>P42YT6M</duree>
5 |
6 | <!-- -2 heures -->
7 | <duree>-PT2H</duree>
8 |
9 | <!-- 2 jours -->
10 | <duree>P2D</duree>
11 | <!-- 10.5 secondes -->
12 | <duree>PT10.5S</duree>
```

Le type date

Le type **date** permet d'exprimer une date. A l'image du type **duration**, une date s'exprime selon une expression bien spécifique à savoir YYYY-MM-DD.

Une nouvelle fois, je vous propose de décortiquer tout ça :

- YYYY représente l'année (year) sur 4 chiffres ou plus.
- MM représente le mois (month) sur 2 chiffres.
- DD représente le jour (day) également sur 2 chiffres.

Dans le cas où l'on souhaite exprimer une date avant Jésus-Christ, un signe - peut-être placé devant l'expression.

Voyons ensemble quelques exemples :

```
1 | <xsd:element name="madate" type="xsd:date" />
2 | 
3 | 1 <!-- 13 janvier 1924 -->
4 | <madate>1924-01-13</madate>
5 | 
6 | 2 <!-- 12 décembre 34 avant JC -->
7 | <madate>-0034-12-12</madate>
8 | 
9 | 3 <!-- 4 novembre 12405 -->
10| <madate>12405-11-04</madate>
```

Le type time

Le type **time** permet d'exprimer une heure. Encore une fois, une expression bien spécifique doit être respectée : hh :mm :ss.

Pour continuer avec nos bonnes habitudes, décortiquons ensemble cette expression :

- hh représente les heures (hour) sur 2 chiffres.
- mm représente les minutes (minute) sur 2 chiffres.
- ss représente les secondes (second) sur 2 chiffres entiers ou à virgule.

Voici quelques exemples :

```
1 | <xsd:element name="monheure" type="xsd:time" />
2 | 
3 | 1 <!-- 10 heures et 24 minutes -->
4 | <monheure>10:24:00</monheure>
5 | 
6 | 2 <!-- 2,5 secondes -->
7 | <monheure>00:00:02.5</monheure>
```

Le type dateTime

Le type **dateTime** peut être considéré comme un mélange entre le type **date** et le type **time**. Ce nouveau type permet donc de représenter une date ET une heure. Une nouvelle fois, une expression particulière doit être respectée : YYYY-MM-DDThh :mm :ss.

Je ne vais pas spécifiquement revenir sur cette expression. En effet, comme vous pouvez le constater, il s'agit des expressions du type **date** et du type **time** séparées par la lettre **T**. Je vous laisse vous référer aux types **date** et **time** pour les règles à appliquer.

Le type gYear

Le type **gYear** représente une année sur 4 chiffres ou plus. Dans le cas où l'on souhaite exprimer une année avant Jésus-Christ, un signe - peut-être placé devant l'expression.

Le type gYearMonth

Le type **gYearMonth** représente une année et un mois. Comme pour tous les types que nous venons de voir dans ce chapitre, le type **gYearMonth** doit respecter une expression particulière : YYYY-MM.

Vous l'aurez compris, les règles sont toujours les mêmes. Je vous laisse donc vous reporter au paragraphe traitant du type date pour plus d'informations.

Le type gMonth

Le type **gMonth** représente un mois sur 2 chiffres précédés du symbole –.

Non, ce n'est pas une erreur de frappe, le symbole est bien –. Voyons un exemple :

```
1 | <xsd:element name="mois" type="xsd:gMonth" />
```

```
1 | <!-- mars -->
2 | <mois>--03</mois>
3 |
4 | <!-- décembre -->
5 | <mois>--12</mois>
```

Le type gMonthDay

Le type **gMonthDay** représente un mois et un jour. Une nouvelle fois, une expression particulière doit être utilisée afin d'exprimer ce nouveau type : –MM-DD.

Une nouvelle fois, les règles sont les mêmes que celles que nous avons déjà utilisé précédemment notamment pour le type **date** et le type **gYearMonth**.

Le type gDay

Finalement, nous allons terminer ce chapitre avec le type **gDay** qui représente un jour sur 2 chiffres précédés du symbole —.

Afin de terminer ce chapitre en beauté, voici quelques exemples :

```
1 | <xsd:element name="journee" type="xsd:gDay" />
```

```
1 | <!-- le troisième jour du mois -->
2 | <journee>---03</journee>
3 |
4 | <!-- le douzième jour du mois -->
5 | <journee>---12</journee>
```

Les types numériques

Le tableau récapitulatif

Type	Description	Commentaire
float	représente un nombre flottant sur 32 bits conforme à la norme IEEE 754	
double	représente un nombre flottant sur 64 bits conforme à la norme IEEE 754	
decimal	représente une nombre décimal	
integer	représente un nombre entier	basé sur le type decimal
long	représente un nombre entier	basé sur le type integer
int	représente un nombre entier	basé sur le type long
short	représente un nombre entier	basé sur le type int
byte	représente un nombre entier	basé sur le type short
nonPositiveInteger	représente un nombre entier non positif	basé sur le type integer
negativeInteger	représente un nombre entier négatif	basé sur le type nonPositiveInteger
nonNegativeInteger	représente un nombre entier non négatif	basé sur le type integer
positiveInteger	représente un nombre entier positif	basé sur le type nonNegativeInteger
unsignedLong	représente un nombre entier positif	basé sur le type nonNegativeInteger
unsignedInt	représente un nombre entier positif	basé sur le type unsigned-Long
unsignedShort	représente un nombre entier positif	basé sur le type unsigned-Int
unsignedByte	représente un nombre entier positif	basé sur le type unsigned-Short



Comme bien souvent en informatique, il convient d'écrire les nombres décimaux avec un point et non une virgule. Par exemple 4.2, 5.23, etc.

Plus en détails

Le type float

Comme vous avez déjà pu le lire dans le tableau récapitulatif, le type **float** représente un nombre flottant sur 32 bits et conforme à la norme IEEE 754. Je suis parfaitement conscient que cette définition est incompréhensible pour la plupart des gens, c'est pourquoi nous allons grandement la simplifier.

Le type **float** a été emprunté aux langages de programmation comme le langage C et est encore aujourd'hui utilisé dans des langages plus récents comme Java ou C#. Il représente un **nombre flottant**, c'est-à-dire un nombre entier ou décimal, se trouvant entre les valeurs 3.4×10^{-38} et 3.4×10^{38} .

A cette plage de valeurs, 3 autres peuvent être ajoutées :

- -INF pour moins l'infini.
- +INF pour plus l'infini.
- NaN pour Not a Number, c'est-à-dire pour désigner une valeur non numérique.

Il est tout à fait possible d'écrire un nombre de type **float** avec des exposants. Il convient alors d'utiliser la notation **E** ou **e**.

Pour mieux comprendre toutes ces règles, je vous propose de regarder ensemble quelques exemples :

```

1 | <xsd:element name="nombre" type="xsd:float" />
2 |
3 | <nombre>42</nombre>
4 | <nombre>-42.25</nombre>
5 | <nombre>3E4</nombre>
6 | <nombre>10e-5</nombre>
```

Le type double

Le type **double** est très proche du type **float**, si ce n'est qu'il représente un nombre flottant sur 64 bits et conforme à la norme IEEE 754 au lieu des 32 bits du type **float**. Concrètement, cette différence se traduit par le fait qu'un nombre de type **double** se trouvant entre les valeurs 1.7×10^{-308} et 1.7×10^{308} .

Comme pour le type **float**, les 3 valeurs suivantes peuvent être ajoutées à la liste :

- -INF pour moins l'infini.
- +INF pour plus l'infini.
- NaN pour Not a Number, c'est-à-dire pour désigner une valeur non numérique.

On retrouve également la règle de l'exposant. Je vous laisse vous référer à la définition du type **float** pour plus de détails.

Le type decimal

Comme son nom le laisse deviner, le type **decimal** représente un nombre décimal, c'est-à-dire un nombre qui peut-être entier ou à virgule. Ce nombre peut-être positif ou négatif et donc être précédé du symbole + ou -. Dans le cas d'un nombre où la partie entière est égale à zéro, il n'est pas obligatoire de l'écrire.

Voyons tout de suite quelques exemples afin d'illustrer cette définition :

```
1 | <xsd:element name="nombre" type="xsd:decimal" />
2 |
3 | <nombre>42</nombre>
4 | <nombre>-42.25</nombre>
5 | <nombre>+.42</nombre>
6 | <nombre>00042.420000</nombre>
```

Le type integer

Le type **integer** est basé sur le type **decimal** et représente un nombre entier, c'est-à-dire un nombre sans virgule. Comme pour le type **décimal**, un nombre de type **integer** peut être précédé par le symbole + ou -.

Le type long

Le type **long** est basé sur le type **integer** si ce n'est qu'un nombre de type **long** doit forcément être compris entre les valeurs -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807.

Le type int

Le type **int** est basé sur le type **long** si ce n'est qu'un nombre de type **int** doit forcément être compris entre les valeurs -2 147 483 648 et 2 147 483 647.

Le type short

Le type **short** est basé sur le type **int** si ce n'est qu'un nombre de type **short** doit forcément être compris entre les valeurs -32 768 et 32 768.

Le type byte

Le type **byte** est basé sur le type **short** si ce n'est qu'un nombre de type **byte** doit forcément être compris entre les valeurs -128 et 127.

Le type nonPositiveInteger

Basé sur le type **integer**, le type **nonPositiveInteger** représente un nombre entier qui n'est pas positif. Concrètement, cela correspond à un nombre négatif ou au nombre zéro.

Voyons ensemble un exemple :

```
1 | <xsd:element name="nombre" type="xsd:nonPositiveInteger" />
2 |
3 | <nombre>-42</nombre>
4 | <nombre>0</nombre>
5 | <nombre>-00042</nombre>
```

Le type negativeInteger

Basé sur le type **nonPositiveInteger**, le type **negativeInteger** représente un nombre entier strictement négatif, c'est-à-dire strictement inférieur à zéro.

Le type nonNegativeInteger

Basé sur le type **integer**, le type **nonNegativeInteger** représente un nombre entier qui n'est pas négatif, c'est-à-dire un nombre supérieur ou égal à zéro.

Soit l'exemple suivant :

```
1 | <xsd:element name="nombre" type="xsd:nonPositiveInteger" />
2 |
3 | <nombre>42</nombre>
4 | <nombre>0</nombre>
5 | <nombre>+00042</nombre>
```

Le type positiveInteger

Basé sur le type **nonNegativeInteger**, le type **positiveInteger** représente un nombre entier strictement positif, c'est-à-dire strictement supérieur à zéro.

Le type unsignedLong

Le type **unsignedLong** est basé sur le type **nonNegativeInteger** et représente un entier compris entre les valeurs 0 et 18 446 744 073 709 551 615.

Le type **unsignedInt**

Le type **unsignedInt** est basé sur le type **unsignedLong** et représente un entier compris entre les valeurs 0 et 4 294 967 295.

Le type **unsignedShort**

Le type **unsignedShort** est basé sur le type **unsignedInt** et représente un entier compris entre les valeurs 0 et 65 535.

Le type **unsignedByte**

Le type **unsignedByte** est basé sur le type **unsignedShort** et représente un entier compris entre les valeurs 0 et 255.

Les autres types

Le tableau récapitulatif

Type	Description
boolean	représente l'état vrai ou faux
QName	représente un nom qualifié
NOTATION	représente une notation
anyURI	représente une URI
base64Binary	représente une donnée binaire au format Base64
hexBinary	représente une donnée binaire au format hexadecimal

Plus en détails

Le type **boolean**

Le type **boolean**, comme son nom le laisse deviner, représente un **booléen**. Pour ceux qui ignorent de quoi il s'agit, un booléen permet d'indiquer l'un des 2 états suivant : vrai ou faux.

Si 2 états sont possibles, 4 valeurs sont en réalité acceptées :

- true qui représente l'état vrai.
- false qui représente l'état faux.
- 1 qui représente l'état vrai.
- 0 qui représente l'état faux.

Conscient que cette notion n'est pas forcément facile à comprendre lorsque c'est la première fois qu'on la rencontre, je vais tenter de l'illustrer avec un exemple. Notre

exemple va nous permettre, via un attribut, de savoir si une personne est un animal ou pas.

```

1 | <xsd:attribute name="animal" type="xsd:boolean" />
2 |
3 | <!-- Victor Hugo n'est pas un animal -->
4 | <personne animal="false">Victor Hugo</personne>
5 | <!-- Zozor est bien un animal -->
6 | <personne animal="true">Zozor</personne>
```

Le type QName

Le type **QName** représente un **nom qualifié**. C'est un concept qui s'appuie sur l'utilisation des espaces de nom.

Le type NOTATION

Le type **NOTATION** permet d'identifier et décrire du contenu XML ou non comme par exemple une image.

Le type anyURI

Comme son nom l'indique, le type **anyURI** représente une **URI** (Uniform Resource Identifier). Une URI est une chaîne de caractère qui permet d'identifier une ressource.

On distingue généralement 2 types d'URI :

- Les **URL** (Uniform Resource Locator) est probablement la forme d'URI la plus connue et je suis sûr que ce nom vous parle. En effet, les URL sont utilisés pour décrire l'adresse d'une ressource sur un réseau. Par exemple `http://www.monsite.com` et `ftp://ftp.monftp.com/notes.txt` sont deux URL possibles.
- Les **URN** (Uniform Resource Name) sont quant à eux utilisés pour identifier une ressource dans un espace de noms. Je ne vais pas m'attarder sur les URN car nous reviendrons plus tard dans ce cours sur la notion des espaces de nom.

Une URI permet d'identifier une ressource de manière relative ou absolue.

Voyons ensemble quelques exemples :

```

1 | <xsd:attribute name="adresse" type="xsd:anyURI"/>
2 |
3 | <!-- URI absolu -->
4 | <image adresse="http://www.siteduzero.com/bundles/common/images/
5 |   /spreadsheetV32.png" />
6 |
7 | <!-- URI relatif -->
8 | <image adresse="../bundles/common/images/spreadsheetV32.png"/>
```

Le type **base64Binary**

Le type **base64Binary** représente une donnée binaire au format Base64.

Comme de nombreux types que nous avons vu, le type base64Binary impose le respect de plusieurs règles :

- Seules les lettres (majuscules ou minuscules), les chiffres et les symboles + / et = sont autorisés.
- Le nombre de caractères qui composent la chaîne doit être un multiple de 4.

Dans le cas où le symbole = est utilisé, de nouvelles règles doivent être respectées :

- Il ne peut apparaître qu'en fin de chaîne, une fois ou deux.
- Dans le cas où il est utilisé qu'une seule fois, il doit forcément être précédé des caractères A Q g ou w.
- Dans le cas où il est utilisé 2 fois, il doit forcément être précédé des caractères A E I M Q U Y c g k o s w 0 (zéro) 4 ou 8.

Le type **hexBinary**

Le type **hexBinary** représente une donnée binaire au format hexadecimal.

Si comme pour le type base64Binary quelques règles sont à respecter, elles sont bien plus simples dans le cas du type **hexBinary**. Ainsi, seuls les lettres entre A et F (majuscules ou minuscules), ainsi que les chiffres sont autorisés. A noter : le nombre de caractères composant la chaîne doit forcément être un multiple de 2.

En résumé

Les 4 grandes familles de types simples permettent d'être très précis quant à la description des éléments simples et des attributs.

Chapitre 13

Schéma XML : les types complexes

Difficulté : 

Dans les chapitres précédents, nous avons vu les **éléments simples** et les différentes familles de types simples. Malheureusement, toutes ces connaissances ne sont pas suffisantes si l'on souhaite pouvoir décrire toutes les structures que les documents XML offrent.

Pour pallier ce manque, nous allons maintenant aborder la suite du cours via l'étude d'un nouveau type d'élément : les **éléments complexes**.



Définition

Bref rappel

Au cours des chapitres précédents, nous avons vu ensemble ce qu'est un **élément simple**, à savoir un élément qui ne contient qu'une valeur dont le type est dit simple. Un élément simple ne contient pas d'autres éléments ni aucun attribut.

Nous avons également vu comment déclarer un élément simple ainsi qu'un attribut. Cependant nous n'avons pas vu comment déclarer un attribut dans un élément. En effet, un élément qui possède un attribut n'est plus un élément simple. On parle alors d'**élément complexe**.

Les éléments complexes

Un **élément complexe** est un élément qui contient d'autres éléments ou des attributs. Bien évidemment les éléments contenus dans un éléments peuvent également contenir des éléments ou des attributs. J'espère que vous suivez toujours !

Je vous propose de voir quelques exemples d'éléments XML qui dans un Schéma XML sont considérés comme complexes :

```
1 <!-- la balise personne contient d'autres balises => élément
2   complexe -->
3 <personne>
4   <!-- la balise nom est un élément simple -->
5   <nom>ROBERT</nom>
6   <!-- la balise prenom est un élément simple -->
7   <prenom>Axel</prenom>
8 </personne>
9 <!-- la balise personne possède un attribut => élément complexe
10  -->
11 <personne sexe="feminin">Axel ROBERT</personne>
```

Abordons maintenant différents exemples qui vont nous permettre de voir et de comprendre comment déclarer des éléments complexes dans un Schéma XML.

Déclarer un élément complexe

Si vous souhaitez déclarer une balise en tant qu'élément complexe, c'est le mot clef **complexType** qu'il faut utiliser associé à celui que nous connaissons déjà : **element**. N'oubliez pas de précéder son utilisation par **xsd** :

```
1 <xsd:element name="mon_nom">
2   <xsd:complexType>
3     <!-- contenu ici -->
4   <xsd:complexType>
5 </xsd:element>
```

Nous reviendrons juste après sur la notion de **contenu**, ne vous inquiétez pas. Reprenons l'un des éléments de type complexe que nous avons vu un peu plus haut :

```
1 | <personne>
2 |   <nom>ROBERT</nom>
3 |   <prenom>Axel</prenom>
4 | </personne>
```

Voici comment le déclarer :

```
1 | <xsd:element name="personne">
2 |   <xsd:complexType>
3 |     <!-- contenu ici -->
4 |   </xsd:complexType>
5 | </xsd:element>
```

Les contenus des types complexes

Concernant les types complexes, il est important de noter qu'il existe 3 types de contenus possibles :

- Les contenus **simples**.
- Les contenus « **standards** ».
- Les contenus **mixtes**.

Les contenus simples

Définition

Le premier type de contenu possible pour un élément complexe est le **contenu simple**.

On appelle **contenu simple**, le contenu d'un élément complexe qui n'est composé que d'attributs et d'un texte de type simple.

Quelques exemples

Je vous propose de voir quelques exemples d'éléments complexes dont le contenu est dit simple.

```
1 | <!-- contient un attribut et du texte -->
2 | <prix devise="euros">35</prix>
3 |
4 | <!-- contient un attribut et du texte -->
5 | <voiture marque="Renault">Clio</voiture>
```

Du côté du Schéma XML

La syntaxe

Pour déclarer un élément complexe faisant référence à une balise contenant des attributs et du texte, voici la syntaxe à utiliser :

```
1 | <xsd:element name="mon_nom">
2 |   <xsd:complexType>
3 |     <xsd:simpleContent>
4 |       <xsd:extension base="mon_type">
5 |         <xsd:attribute name="mon_nom" type="mon_type" />
6 |       </xsd:extension>
7 |     </xsd:simpleContent>
8 |   </xsd:complexType>
9 | </xsd:element>
```

Un exemple

Reprenons l'exemple d'un prix prenant pour attribut une devise :

```
1 | <prix devise="euros">35</prix>
```

Voici alors le schéma XML associé :

```
1 | <xsd:element name="prix">
2 |   <xsd:complexType>
3 |     <xsd:simpleContent>
4 |       <xsd:extension base="xsd:positiveInteger">
5 |         <xsd:attribute name="devise" type="xsd:string" />
6 |       </xsd:extension>
7 |     </xsd:simpleContent>
8 |   </xsd:complexType>
9 | </xsd:element>
```

Dans le cas où la balise que l'on cherche à décrire contient plusieurs attributs, il convient de tout simplement les lister entre les balises `<xsd:extension>`. Par exemple :

```
1 | <voiture marque="Renault" type="essence">Clio</voiture>
2 |
3 | <xsd:element name="voiture">
4 |   <xsd:complexType>
5 |     <xsd:simpleContent>
6 |       <xsd:extension base="xsd:string">
7 |         <xsd:attribut name="marque" type="xsd:string" />
8 |         <xsd:attribut name="type" type="xsd:string" />
9 |       </xsd:extension>
```

```

8 |         <xsd:simpleContent>
9 |             </xsd:complexType>
10|        </xsd:element>

```

Comme vous pouvez le constater, on se contente de mettre à la suite les différents attributs qui composent l'élément. À noter que l'ordre dans lequel les attributs sont déclarés dans le Schéma XML n'a aucune importance.

Les contenus "standards"

Définition

Après les contenus simples, nous allons monter la barre d'un cran et nous attaquer aux **contenus « standards »**.



Il est important de noter que cette appellation n'est nullement officielle. C'est une appellation maison car il s'agit du cas de figure qui à tendance à revenir le plus souvent.

Ce que j'appelle **contenu « standard »**, c'est le contenu d'un élément complexe qui n'est composé que d'autres éléments (simples ou complexes) ou uniquement d'attributs.

Quelques exemples

Comme pour le contenu simple, voyons quelques exemples de contenu « standard » :

```

1 | <!-- contient d'autres éléments -->
2 | <personne>
3 |     <nom>DUPONT</nom>
4 |     <prenom>Robert</prenom>
5 | </prenom>
6 |
7 | <!-- contient un attribut -->
8 | <voiture marque="Renault" />

```

Balise contenant un ou plusieurs attributs

Je vous propose de débuter par le cas de figure le plus simple, à savoir celui d'un élément complexe qui ne contient que des attributs.

Reprenons l'exemple de notre voiture du dessus :

```
1 | <voiture marque="Renault" />
```

Voici alors le Schéma XML associé :

```
1 | <xsd:element name="voiture">
2 |   <xsd:complexType>
3 |     <xsd:attribut name="marque" type="xsd:string" />
4 |   </xsd:complexType>
5 | </xsd:element>
```

Il n'y a, pour le moment, rien de bien compliqué. On se contente d'imbriquer une balise `<xsd:attribut />` dans une balise `<xsd:complexType />`.

Si l'on tente de complexifier un petit peu les choses, nous allons nous rendre compte que, dans le fond, rien ne change. Prenons par exemple le cas d'une balise contenant plusieurs attributs :

```
1 | <voiture marque="Renault" modele="Clio" />
```

Regardons alors le Schéma XML :

```
1 | <xsd:element name="voiture">
2 |   <xsd:complexType>
3 |     <xsd:attribut name="marque" type="xsd:string" />
4 |     <xsd:attribut name="modele" type="xsd:string" />
5 |   </xsd:complexType>
6 | </xsd:element>
```

Comme vous pouvez le constater, on se contente de mettre à la suite les différents attributs qui composent l'élément. Une fois de plus, l'ordre dans lequel les balises `<xsd:attribut />` sont placées n'a aucune importance.

Balise contenant d'autres éléments

Il est maintenant temps de passer à la suite et de jeter un coup d'œil aux balises qui contiennent d'autres éléments.

La séquence

Une **séquence** est utilisée lorsque l'on souhaite spécifier que les éléments contenus dans un type complexe doivent apparaître dans un ordre précis.

Voici comment se déclare une **séquence** au niveau d'un Schéma XML :

```
1 | <xsd:element name="mon_nom">
2 |   <xsd:complexType>
3 |     <xsd:sequence>
4 |       <!-- liste des éléments -->
5 |     </xsd:sequence>
6 |     <!-- listes des attributs -->
7 |   </xsd:complexType>
8 | </xsd:element>
```

Voyons tout de suite un exemple :

```

1 <xsd:element name="personne">
2   <xsd:complexType>
3     <xsd:sequence>
4       <xsd:element name="nom" type="xsd:string"/>
5       <xsd:element name="prenom" type="xsd:string"/>
6     </xsd:sequence>
7     <xsd:attribute name="sexe" type="xsd:string" />
8   </xsd:complexType>
9 </xsd:element>

```

Cet extrait signifie que la balise `<personne />` qui possède l'attribut `sexe`, contient les balises `<nom />` et `<prenom />` dans cet ordre.

Illustrons alors cet exemple :

```

1 <!-- valide -->
2 <personne sexe="masculin">
3   <nom>DUPONT</nom>
4   <prenom>Robert</prenom>
5 </personne>
6
7 <!-- invalide => les balises nom et prenom sont inversées -->
8 <personne sexe="masculin">
9   <prenom>Robert</prenom>
10  <nom>DUPONT</nom>
11 </personne>

```

Le type all

Le type **all** est utilisé lorsque l'on veut spécifier que les éléments contenu dans un type complexe peuvent apparaître dans n'importe quel ordre. Ils doivent cependant tous apparaître une et une seule fois.

Voici comment se déclare le type **all** au niveau d'un Schéma XML :

```

1 <xsd:element name="mon_nom">
2   <xsd:complexType>
3     <xsd:all>
4       <!-- liste des éléments -->
5     </xsd:all>
6       <!-- listes des attributs -->
7   </xsd:complexType>
8 </xsd:element>

```

Voyons tout de suite un exemple :

```

1 <xsd:element name="personne">
2   <xsd:complexType>
3     <xsd:all>
4       <xsd:element name="nom" type="xsd:string"/>
5       <xsd:element name="prenom" type="xsd:string"/>

```

```
6      </xsd:all>
7  </xsd:complexType>
8 </xsd:element>
```

Cet extrait signifie donc que la balise `<personne />` contient les balises `<nom />` et `<prenom />` dans n'importe quel ordre.

Illustrons alors cet exemple :

```
1 <!-- valide -->
2 <personne sexe="masculin">
3   <nom>DUPONT</nom>
4   <prenom>Robert</prenom>
5 </prenom>
6
7 <!-- valide -->
8 <personne sexe="masculin">
9   <prenom>Robert</prenom>
10  <nom>DUPONT</nom>
11 </prenom>
```

Le choix

Un **choix** est utilisé lorsque l'on veut spécifier qu'un élément contenu dans un type complexe soit choisi dans une liste pré-définie.

Voici comment se déclare un **choix** au niveau d'un Schéma XML :

```
1 <xsd:element name="mon_nom">
2   <xsd:complexType >
3     <xsd:choice>
4       <!-- liste des éléments -->
5     </xsd:choice>
6     <!-- listes des attributs -->
7   </xsd:complexType>
8 </xsd:element>
```

Voyons sans plus tarder un exemple :

```
1 <xsd:element name="personne">
2   <xsd:complexType>
3     <xsd:choice>
4       <xsd:element name="nom" type="xsd:string"/>
5       <xsd:element name="prenom" type="xsd:string"/>
6     </xsd:choice>
7   </xsd:complexType>
8 </xsd:element>
```

Cet extrait signifie donc que la balise `<personne />` contient soit la balise `<nom />`, soit `<prenom />`.

Illustrons cet exemple :

```

1 <!-- valide -->
2 <personne sexe="masculin">
3   <nom>DUPONT</nom>
4 </prenom>
5
6 <!-- valide -->
7 <personne sexe="masculin">
8   <prenom>Robert</prenom>
9 </prenom>
10
11 <!-- invalide => les 2 balises prenom et nom ne peuvent pas
     apparaître en même temps -->
12 <personne sexe="masculin">
13   <prenom>Robert</prenom>
14   <nom>DUPONT</nom>
15 </prenom>
```

Cas d'un type complexe encapsulant un type complexe

Avant de terminer cette partie, il nous reste un cas à voir : celui d'un type complexe encapsulant également un type complexe.

Prenons par exemple le document XML suivant :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <personne>
3   <identite>
4     <nom>NORRIS</nom>
5     <prenom>Chuck</prenom>
6   </identite>
7 </personne>
```

Ce document XML permet d'identifier une personne via son nom et son prénom. Voyons alors le Schéma XML qui définit notre document XML :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <xsd:element name="personne">
4     <xsd:complexType>
5       <xsd:sequence>
6         <xsd:element name="identite">
7           <xsd:complexType>
8             <xsd:sequence>
9               <xsd:element name="nom" type="xsd:string"/>
10              <xsd:element name="prenom" type="xsd:string"/>
11            </xsd:sequence>
12          </xsd:complexType>
13        </xsd:element>
```

```
14 |         </xsd:sequence>
15 |     </xsd:complexType>
16 |     </xsd:element>
17 | </xsd:schema>
```

En soit, il n'y a rien de compliqué. Il convient juste de repérer que lorsque l'on place un élément complexe au sein d'un autre élément complexe, dans notre cas, une **identité** dans une **personne**, il convient d'utiliser une **séquence**, un **choix** ou un type **all**.

Définition

Il est temps de conclure ce chapitre avec le dernier type de contenu possible : les **contenus mixtes**.

Un **contenu mixte** est le contenu d'un élément complexe qui est composé d'attributs, d'éléments et de texte.

Un exemple

Afin d'illustrer cette définition, je vous propose de nous appuyer sur un exemple :

```
1 | <balise1>
2 |   Ceci est une chaîne de caractères
3 |   <balise2>10</balise2>
4 |   7.5
5 | </balise1>
```

Du côté du Schéma XML

La syntaxe

Pour déclarer un élément complexe au contenu mixte, voici la syntaxe à utiliser :

```
1 | <xsd:element name="mon_nom">
2 |   <xsd:complexType mixed="true">
3 |     <!-- liste des éléments -->
4 |     </xsd:complexType>
5 |     <!-- liste des attributs -->
6 |   </xsd:element>
```

La nouveauté est donc l'utilisation du mot clef **mixed**.

Un exemple

Prenons l'exemple d'une facture fictive dans laquelle on souhaite identifier l'acheteur et la somme qu'il doit payer.

```
1 | <facture><acheteur>Zozor</acheteur>, doit payer <somme>1000</
   |     somme> euros.</facture>
```

Voici comment le traduire au sein d'un Schéma XML :

```
1 | <xsd:element name="facture">
2 |   <xsd:complexType mixed="true">
3 |     <xsd:sequence>
4 |       <xsd:element name="acheteur" type="xsd:string" />
5 |       <xsd:element name="somme" type="xsd:int" />
6 |     </xsd:sequence>
7 |   </xsd:complexType>
8 | </xsd:element>
```

Comme vous pouvez le remarquer, j'ai utilisé la balise `<xsd:sequence />` pour encapsuler la liste des balises contenues dans la balise `<facture />`, mais vous pouvez bien évidemment adapter à votre cas de figure et choisir parmi les balises que nous avons vu dans le chapitre précédent, à savoir :

- `<xsd:sequence />`.
- `<xsd:all />`.
- `<xsd:choice />`.

En résumé

- Un **élément complexe** est un élément qui contient d'autres éléments ou des attributs.
- Un **élément complexe** est décrit grâce à la balise `<xsd:complexType />`.
- Un **élément complexe** a 3 types de contenus possibles : les **contenus simples**, « **standards** » et **mixtes**.

Chapitre 14

Schéma XML : aller plus loin

Difficulté : 

Le prochain chapitre est l'occasion de terminer notre apprentissage des **Schémas XML**. Ce chapitre regroupe de nombreuses notions qui vous seront utiles pour décrire tous les documents XML possibles et imaginables.

Ce chapitre s'annonce très dense, accrochez-vous, le résultat en vaut la peine ! Au programme : la **gestion du nombre d'occurrences** d'un élément, la **réutilisation des éléments**, l'**héritage**, les **identifiants** et pour terminer **EditiX** !



Le nombre d'occurrences

Dans le chapitre précédent, nous avons vu comment écrire des éléments de type complexe. Je peux maintenant vous avouer que je vous ai caché quelques petites choses. Nous allons les découvrir ensemble dès à présent.

La première concerne le **nombre d'occurrences d'une balise**. Pour vous aider à bien comprendre cette notion, je vous propose d'étudier un morceau de Schéma XML que nous avons déjà vu. Il s'agit de celui d'une personne qui possède un nom et un prénom :

```
1 | <xsd:complexType name="personne">
2 |   <xsd:sequence>
3 |     <xsd:element name="nom" type="xsd:string"/>
4 |     <xsd:element name="prenom" type="xsd:string"/>
5 |   </xsd:sequence>
6 | </xsd:complexType>
```

Comme je vous le disais précédemment, cet extrait signifie que la balise `<personne />` contient les balises `<nom />` et `<prenom />` dans cet ordre.

La notion d'**occurrence** va nous permettre de préciser si les balises, dans le cas de notre exemple `<nom />` et `<prenom />`, peuvent apparaître plusieurs fois, voire pas du tout.

Le cas par défaut

Le cas par défaut est celui que nous avons vu jusqu'à maintenant. Lorsque le nombre d'occurrences n'est pas précisé, la balise doit apparaître une et une seule fois.

Le nombre minimum d'occurrences

Pour indiquer le nombre minimum d'occurrences d'un élément, on utilise l'attribut **minOccurs**. Comme nous l'avons déjà vu plus haut, sa valeur par défaut est 1. A noter : dans le cas où il est utilisé, sa valeur doit obligatoirement être supérieure à zéro.

Le nombre maximum d'occurrences

Pour indiquer le nombre maximum d'occurrences d'un élément, on utilise l'attribut **maxOccurs**. Comme pour le nombre minimum d'occurrences, la valeur par défaut est 1. Une nouvelle fois, dans le cas où il est utilisé, sa valeur doit obligatoirement être supérieure à zéro. A noter : il est également possible de ne pas spécifier un nombre maximal d'occurrences grâce au mot clef **unbounded**.

Exemple

Je vous propose de terminer ce chapitre en l'illustrant par un exemple.

```
1 <xsd:complexType name="personne">
2   <xsd:sequence>
3     <xsd:element name="nom" type="xsd:string" />
4     <xsd:element name="prenom" type="xsd:string" minOccurs=
5       "2" maxOccurs="unbounded" />
6   </xsd:sequence>
7 </xsd:complexType>
```

Dans l'extrait de Schéma XML ci-dessus, on remarque que pour l'élément `prenom`, le nombre minimum d'occurrences est à 2 tandis qu'il n'y a pas de maximum. Cela signifie, que dans notre fichier XML, cette balise devra apparaître entre 2 et une infinité de fois comme en témoigne les extraits de fichier XML suivants :

```
1 <personne>
2   <nom>Zozor</nom>
3   <prenom>Robert</prenom>
4   <prenom>Bernard</prenom>
5 </personne>
6
7 <personne>
8   <nom>Zozor</nom>
9   <prenom>Robert</prenom>
10  <prenom>Bernard</prenom>
11  <prenom>Paul</prenom>
12  <prenom>Pierre</prenom>
13 </personne>
```

La réutilisation des éléments

Avant d'attaquer la notion d'**héritage**, nous allons voir comment réutiliser des **types complexes** afin d'en écrire le moins possible. C'est bien connu, les informaticiens sont des fainéants !

Pourquoi ne pas tout écrire d'un seul bloc ?

Puisqu'un exemple est souvent bien plus parlant que de longues explications, je vous propose d'étudier le document XML suivant :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <banque>
3   <!-- 1er client de la banque -->
4   <client>
5     <!-- identité du client -->
6     <identite>
```

```
7      <nom>NORRIS</nom>
8      <prenom>Chuck</prenom>
9    </identite>
10
11    <!-- liste des comptes bancaires du client -->
12    <comptes>
13      <livretA>
14        <montant>2500</montant>
15      </livretA>
16      <courant>
17        <montant>4000</montant>
18      </courant>
19    </comptes>
20  </client>
21 </banque>
```

Ce document XML représente une banque et ses clients. Pour chaque client, on connaît son identité, le montant de son livret A ainsi que le montant de son compte courant.

Avec nos connaissances actuelles, voici ce à quoi ressemble le Schéma XML qui décrit ce document XML :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3    <xsd:element name="banque">
4      <xsd:complexType>
5        <xsd:sequence>
6          <xsd:element name="client">
7            <xsd:complexType>
8              <xsd:sequence>
9                <xsd:element name="identite" maxOccurs="unbounded"
10                  >
11                  <xsd:complexType>
12                    <xsd:sequence>
13                      <xsd:element name="nom" type="xsd:string" /
14                        >
15                      <xsd:element name="prenom" type="xsd:string"
16                        " />
17                    </xsd:sequence>
18                  </xsd:complexType>
19                </xsd:element>
20                <xsd:element name="comptes">
21                  <xsd:complexType>
22                    <xsd:sequence>
23                      <xsd:element name="livretA">
24                        <xsd:complexType>
25                          <xsd:sequence>
26                            <xsd:element name="montant" type="xsd
27                              : double" />
28                          </xsd:sequence>
29                      </xsd:complexType>
30                    </xsd:sequence>
31                  </xsd:complexType>
32                </xsd:element>
33              </xsd:sequence>
34            </xsd:complexType>
35          </xsd:element>
36        </xsd:sequence>
37      </xsd:complexType>
38    </xsd:element>
39  </xsd:schema>
```

```
26      </xsd:element>
27      <xsd:element name="courant">
28          <xsd:complexType>
29              <xsd:sequence>
30                  <xsd:element name="montant" type="xsd
31                      :double" />
32          </xsd:sequence>
33      </xsd:complexType>
34      </xsd:element>
35          </xsd:sequence>
36      </xsd:complexType>
37      </xsd:element>
38          </xsd:sequence>
39      </xsd:complexType>
40          </xsd:element>
41      </xsd:sequence>
42  </xsd:complexType>
43  </xsd:element>
44 </xsd:schema>
```

Cette construction d'un seul bloc, également appelé **construction « en pouپées russes »** n'est pas des plus lisibles. Afin de rendre notre **Schéma XML** un peu plus clair et compréhensible, je vous propose de le diviser. Sa lecture en sera grandement facilitée.

Diviser un Schéma XML

Quelques explications

Dans cette partie nous allons voir comment « casser » cette écriture « en pouپées russes » afin de rendre notre Schéma XML plus compréhensible et plus accessible.

L'idée est assez simple dans son ensemble. On déclare de manière globale les différents éléments qui composent notre Schéma XML, puis, dans nos structures complexes, on y fait référence.

La syntaxe

La déclaration d'un élément ne change pas par rapport à ce que nous avons vu jusqu'à maintenant, qu'il s'agisse d'un élément simple ou d'un élément complexe.

Établir une référence est alors très simple grâce au mot clef **ref** :

```
1 | <xsd:element ref="mon_nom" />
```

Je vous propose d'illustrer l'utilisation de ce nouveau mot clef *via* un exemple. Nous allons décomposer l'identité d'un client. Pour rappel, voici ce que nous avons écrit dans notre premier essai :

```
1 <xsd:element name="identite" maxOccurs="unbounded" >
2     <xsd:complexType>
3         <xsd:sequence>
4             <xsd:element name="nom" type="xsd:string" />
5             <xsd:element name="prenom" type="xsd:string" />
6         </xsd:sequence>
7     </xsd:complexType>
8 </xsd:element>
```

Nous allons donc faire une déclaration globale des éléments *nom* et *prenom* afin d'y faire référence dans l'élément complexe *identite*.

```
1 <!-- déclaration globale de certains éléments -->
2 <xsd:element name="nom" type="xsd:string" />
3 <xsd:element name="prenom" type="xsd:string" />
4
5 <xsd:element name="identite" maxOccurs="unbounded" >
6     <xsd:complexType>
7         <xsd:sequence>
8             <xsd:element ref="nom" />
9             <xsd:element ref="prenom" />
10            </xsd:sequence>
11        </xsd:complexType>
12 </xsd:element>
```

Comme vous pouvez le constater, l'ensemble est déjà plus lisible. Cette méthode nous permet également de réutiliser des éléments qui reviennent plusieurs fois comme par exemple l'élément *montant* présent dans le compte courant et le livret A d'un client.

Mise à jour de notre banque et ses clients

Mettons alors à jour notre Schéma XML afin de le découper le plus possible :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3     <!-- déclaration des éléments -->
4     <xsd:element name="nom" type="xsd:string" />
5     <xsd:element name="prenom" type="xsd:string" />
6     <xsd:element name="montant" type="xsd:double" />
7
8     <xsd:element name="banque">
9         <xsd:complexType>
10            <xsd:sequence>
11                <xsd:element name="client">
12                    <xsd:complexType>
13                        <xsd:sequence>
14                            <xsd:element name="identite" maxOccurs="unbounded"
15                                >
16                                <xsd:complexType>
17                                    <xsd:sequence>
```

```

17         <xsd:element ref="nom" />
18         <xsd:element ref="prenom" />
19     </xsd:sequence>
20   </xsd:complexType>
21 </xsd:element>
22 <xsd:element name="comptes">
23   <xsd:complexType>
24     <xsd:sequence>
25       <xsd:element name="livretA">
26         <xsd:complexType>
27           <xsd:sequence>
28             <xsd:element ref="montant" />
29           </xsd:sequence>
30         </xsd:complexType>
31       </xsd:element>
32       <xsd:element name="courant">
33         <xsd:complexType>
34           <xsd:sequence>
35             <xsd:element ref="montant" />
36           </xsd:sequence>
37         </xsd:complexType>
38       </xsd:element>
39     </xsd:sequence>
40   </xsd:complexType>
41   </xsd:element>
42 </xsd:sequence>
43 </xsd:complexType>
44 </xsd:element>
45 </xsd:sequence>
46 </xsd:complexType>
47 </xsd:element>
48 </xsd:schema>
```

Comme vous pouvez le constater, c'est mieux, mais selon moi, ce n'est pas encore ça. Plutôt que de déclarer globalement uniquement des éléments simples comme le *montant*, le *nom* ou le *prenom*, pourquoi ne pas déclarer globalement des éléments complexes comme l'identité du client ou encore le livret A ou le compte courant.

Voyons alors le résultat :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <!-- déclaration des éléments -->
4   <xsd:element name="montant" type="xsd:double" />
5
6   <xsd:element name="identite" maxOccurs="unbounded" >
7     <xsd:complexType>
8       <xsd:sequence>
9         <xsd:element name="nom" type="xsd:string" />
10        <xsd:element name="prenom" type="xsd:string" />
11      </xsd:sequence>
```

```
12     </xsd:complexType>
13 </xsd:element>
14
15 <xsd:element name="livretA">
16     <xsd:complexType>
17         <xsd:sequence>
18             <xsd:element ref="montant" />
19         </xsd:sequence>
20     </xsd:complexType>
21 </xsd:element>
22
23 <xsd:element name="courant">
24     <xsd:complexType>
25         <xsd:sequence>
26             <xsd:element ref="montant" />
27         </xsd:sequence>
28     </xsd:complexType>
29 </xsd:element>
30
31 <xsd:element name="comptes">
32     <xsd:complexType>
33         <xsd:sequence>
34             <xsd:element ref="livretA" />
35             <xsd:element ref="courant" />
36         </xsd:sequence>
37     </xsd:complexType>
38 </xsd:element>
39
40 <xsd:element name="client">
41     <xsd:complexType>
42         <xsd:sequence>
43             <xsd:element ref="identite" />
44             <xsd:element ref="comptes" />
45         </xsd:sequence>
46     </xsd:complexType>
47 </xsd:element>
48
49 <!-- Schéma XML -->
50 <xsd:element name="banque">
51     <xsd:complexType>
52         <xsd:sequence>
53             <xsd:element ref="client" />
54         </xsd:sequence>
55     </xsd:complexType>
56 </xsd:element>
57 </xsd:schema>
```

C'est tout de même plus lisible, non ?

Créer ses propres types

Grâce aux **références**, nous sommes arrivés à un résultat satisfaisant, mais si l'on regarde en détail, on se rend vite compte que notre Schéma XML n'est pas optimisé. En effet, les différents comptes de notre client, à savoir le livret A et le compte courant, ont des structures identiques et pourtant, nous les déclarons 2 fois.

Dans cette partie, nous allons donc apprendre à créer nos propres types pour encore et toujours en écrire le moins possible !

La syntaxe

Déclarer un nouveau type, n'est pas plus compliqué que ce que nous avons vu jusqu'à présent. Il est cependant important de noter une petite chose. Les types que nous allons créer peuvent être de 2 natures : **simple** ou **complexe**.

Débutons avec la création et l'utilisation d'un **type simple**. La création d'un **type simple** est utile lorsque par exemple dans un Schéma XML, plusieurs chaînes de caractères ou plusieurs nombres ont des restrictions similaires.

```

1 <!-- création -->
2 <xsd:simpleType name="mon_type_perso">
3   <xsd:restriction base="mon_type">
4     <!-- liste des restrictions -->
5   </xsd:restriction>
6 </xsd:simpleType>
7
8 <!-- utilisation-->
9 <xsd:element name="mon_nom" type="mon_type_perso" />
```

Continuons avec la création et l'utilisation d'un **type complexe** :

```

1 <!-- création -->
2 <xsd:ComplexType name="mon_type_perso">
3   <!-- personnalisation du type complexe -->
4 </xsd:ComplexType>
5
6 <!-- utilisation-->
7 <xsd:element name="mon_nom" type="mon_type_perso" />
```

Mise à jour de notre banque et ses clients

Je vous propose de mettre à jour notre Schéma XML en créant un type « *compte* » que l'on pourra utiliser pour le *livret A* et le *compte courant* des clients de notre banque.

Voici alors ce que ça donne :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <!-- déclaration des éléments -->
```

```
4   <xsd:element name="montant" type="xsd:double" />
5
6   <xsd:element name="identite" >
7       <xsd:complexType>
8           <xsd:sequence>
9               <xsd:element name="nom" type="xsd:string" />
10              <xsd:element name="prenom" type="xsd:string" />
11          </xsd:sequence>
12      </xsd:complexType>
13  </xsd:element>
14
15  <xsd:complexType name="compte">
16      <xsd:sequence>
17          <xsd:element ref="montant" />
18      </xsd:sequence>
19  </xsd:complexType>
20
21  <xsd:element name="comptes">
22      <xsd:complexType>
23          <xsd:sequence>
24              <xsd:element name="livretA" type="compte" />
25              <xsd:element name="courant" type="compte" />
26          </xsd:sequence>
27      </xsd:complexType>
28  </xsd:element>
29
30  <xsd:element name="client">
31      <xsd:complexType>
32          <xsd:sequence>
33              <xsd:element ref="identite" />
34              <xsd:element ref="comptes" />
35          </xsd:sequence>
36      </xsd:complexType>
37  </xsd:element>
38
39  <!-- Schéma XML -->
40  <xsd:element name="banque">
41      <xsd:complexType >
42          <xsd:sequence>
43              <xsd:element ref="client" maxOccurs="unbounded"
44                  />
45          </xsd:sequence>
46      </xsd:complexType>
47  </xsd:element>
48 </xsd:schema>
```

Notre Schéma XML est bien plus lisible maintenant !

L'héritage

L'**héritage** est un concept que l'on retrouve dans la plupart des **langages de programmation orienté objet**. Certes le **XML** n'est pas un langage de programmation, mais ça ne l'empêche pas d'assimiler cette notion.

Pour faire simple, l'**héritage** permet de réutiliser des éléments d'un Schéma XML pour en construire de nouveaux. Si vous avez encore du mal à comprendre le concept, ne vous inquiétez pas, nous allons utiliser des exemples afin de l'illustrer.

En XML, 2 types d'héritages sont possibles :

- Par **restriction**.
- Par **extension**.

Dans les 2 cas, c'est le mot clef **base** que nous utiliserons pour indiquer un héritage.

L'héritage par restriction

La première forme d'héritage que nous allons voir est celle par **restriction**.

Définition

Une **restriction** est une notion qui peut s'appliquer aussi bien aux éléments qu'aux attributs et qui permet de déterminer plus précisément la valeur attendue via la détermination d'un certain nombre de contraintes.

Par exemple, jusqu'à maintenant, nous sommes capables de dire qu'un élément ou qu'un attribut doit contenir un nombre entier strictement positif. Grâce aux **restrictions**, nous allons pouvoir pousser le concept jusqu'au bout en indiquant qu'un élément qu'un attribut doit contenir un nombre entier strictement positif compris entre 1 et 100.

Lorsque l'on déclare une restriction sur un élément, la syntaxe suivante doit être respectée :

```

1 | <xsd:element name="mon_nom">
2 |   <xsd:simpleType>
3 |     <xsd:restriction base="type_de_base">
4 |       <!-- détail de la restriction -->
5 |     </xsd:restriction>
6 |   </xsd:simpleType>
7 | </xsd:element>
```

La syntaxe est quasiment identique dans le cas d'un attribut :

```

1 | <xsd:attribute name="mon_nom">
2 |   <xsd:simpleType>
3 |     <xsd:restriction base="type_de_base">
4 |       <!-- détail de la restriction -->
5 |     </xsd:restriction>
6 |   </xsd:simpleType>
```

7 | </xsd:attribute>

Vous avez très probablement remarqué l'attribut **base** dans la balise `<restriction />`. Il s'agit du type de votre balise et donc de votre restriction. Il suffit donc de choisir un type simple dans la liste de ceux que nous avons vu dans les chapitres précédents (`xsd:string`, `xsd:int`, etc.).

Le tableau récapitulatif

Nom de la restriction	Description
minExclusive	permet de définir une valeur minimale exclusive
minInclusive	permet de définir une valeur minimale inclusive
maxExclusive	permet de définir une valeur maximale exclusive
maxInclusive	permet de définir une valeur maximale inclusive
totalDigits	permet de définir le nombre exact de chiffres qui composent un nombre
fractionDigits	permet de définir le nombre de chiffres autorisés après la virgule
length	permet de définir le nombre exact de caractères d'une chaîne
minLength	permet de définir le nombre minimum de caractères d'une chaîne
maxLength	permet de définir le nombre maximum de caractères d'une chaîne
enumeration	permet d'énumérer la liste des valeurs possibles
whiteSpace	permet de déterminer le comportement à adopter avec les espaces
pattern	permet de définir des expressions rationnelles

Plus en détails



Pour décrire la syntaxe des différentes restrictions, je vais m'appuyer sur la balise `<xsd:element />`, mais n'oubliez pas que toutes les règles que nous voyons dans ce chapitre sont également valables pour la balise `<xsd:attribute />`.

La restriction minExclusive

La restriction **minExclusive** s'applique à un élément de type numérique et permet de définir sa valeur minimale. Comme son nom le laisse deviner, la valeur indiquée est exclue des valeurs que peut prendre l'élément.

Voici sa syntaxe :

1 | <xsd:element name="`mon_nom`">

```

1 <xsd:simpleType>
2     <xsd:restriction base="type_de_base">
3         <xsd:minExclusive value="ma_valeur" />
4     </xsd:restriction>
5 </xsd:simpleType>
6 </xsd:element>
7 
```

Afin d'illustrer cette première restriction, prenons par exemple l'âge d'une personne que l'on souhaite obligatoirement majeure :

```

1 <xsd:complexType name="personne">
2     <xsd:attribute name="age">
3         <xsd:simpleType>
4             <xsd:restriction base="xsd:nonNegativeInteger">
5                 <xsd:minExclusive value="17" />
6             </xsd:restriction>
7         </xsd:simpleType>
8     </xsd:attribute>
9 </xsd:complexType>

1 <!-- valide -->
2 <personne age="18" />
3
4 <!-- valide -->
5 <personne age="43" />
6
7 <!-- invalide -->
8 <personne age="17" />

```

La restriction minInclusive

La restriction **minInclusive** ressemble énormément à la restriction **minExclusive** que nous venons de voir si ce n'est que la valeur indiquée peut-être prise par l'élément.

Voici sa syntaxe :

```

1 <xsd:element name="mon_nom">
2     <xsd:simpleType>
3         <xsd:restriction base="type_de_base">
4             <xsd:minInclusive value="ma_valeur" />
5         </xsd:restriction>
6     </xsd:simpleType>
7 </xsd:element>

```

La restriction maxExclusive

La restriction **maxExclusive** s'applique à un élément de type numérique et permet de définir sa valeur maximale. Comme son nom le laisse deviner, la valeur indiquée est exclue des valeurs que peut prendre l'élément.

Voici sa syntaxe :

```
1 | <xsd:element name="mon_nom">
```

```
1 <xsd:simpleType>
2     <xsd:restriction base="type_de_base">
3         <xsd:maxExclusive value="ma_valeur" />
4     </xsd:restriction>
5 </xsd:simpleType>
6 </xsd:element>
```

La restriction maxInclusive

La restriction **maxInclusive** ressemble énormément à la restriction **maxExclusive** que nous venons de voir si ce n'est que la valeur indiquée peut-être prise par l'élément.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2     <xsd:simpleType>
3         <xsd:restriction base="type_de_base">
4             <xsd:maxInclusive value="ma_valeur" />
5         </xsd:restriction>
6     </xsd:simpleType>
7 </xsd:element>
```

La restriction totalDigits

La restriction **totalDigits** s'applique à un élément de type numérique et permet de définir le nombre exact de chiffres qui composent le nombre. Sa valeur doit obligatoirement être supérieure à zéro.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2     <xsd:simpleType>
3         <xsd:restriction base="type_de_base">
4             <xsd:totalDigits value="ma_valeur" />
5         </xsd:restriction>
6     </xsd:simpleType>
7 </xsd:element>
```

Je vous propose d'illustrer tout de suite cette restriction avec un exemple. Une nouvelle fois prenons l'âge d'une personne. Imaginons un contexte dans lequel l'âge d'une personne doit obligatoirement être compris entre 10 et 99. Il doit donc être obligatoirement composé de 2 chiffres :

```
1 <xsd:complexType name="personne">
2     <xsd:attribute name="age">
3         <xsd:simpleType>
4             <xsd:restriction base="xsd:nonNegativeInteger">
5                 <xsd:totalDigits value="2" />
6             </xsd:restriction>
7         </xsd:simpleType>
8     </xsd:attribute>
9 </xsd:complexType>
```

```

1 | <!-- valide -->
2 | <personne age="18" />
3 |
4 | <!-- valide -->
5 | <personne age="43" />
6 |
7 | <!-- invalide -->
8 | <personne age="4" />
```

La restriction fractionDigits

La restriction **fractionDigits** s'applique à un élément de type numérique et permet de définir le nombre maximal de chiffres qui composent une décimale. Sa valeur peut-être supérieure ou égale à zéro.

Voici sa syntaxe :

```

1 | <xsd:element name="mon_nom">
2 |   <xsd:simpleType>
3 |     <xsd:restriction base="type_de_base">
4 |       <xsd:fractionDigits value="ma_valeur" />
5 |     </xsd:restriction>
6 |   </xsd:simpleType>
7 | </xsd:element>
```

La restriction length

La restriction **length** permet de définir le nombre exact de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

```

1 | <xsd:element name="mon_nom">
2 |   <xsd:simpleType>
3 |     <xsd:restriction base="type_de_base">
4 |       <xsd:length value="ma_valeur" />
5 |     </xsd:restriction>
6 |   </xsd:simpleType>
7 | </xsd:element>
```

Pour illustrer l'utilisation de la La restriction **length**, prenons par exemple une empreinte SHA1. Pour faire simple, une empreinte SHA1 est un nombre hexadécimal composé de 40 caractères :

```

1 | <xsd:complexType name="sha1">
2 |   <xsd:simpleType>
3 |     <xsd:restriction base="xsd:hexBinary">
4 |       <xsd:length value="40" />
5 |     </xsd:restriction>
6 |   </xsd:simpleType>
7 | </xsd:complexType>
```



```

1 | <!-- valide -->
```

```
2 <sha1>edf7a6029d6bdfb68447677a1d76639725f795f1</sha1>
3
4 <!-- valide -->
5 <sha1>a94a8fe5ccb19ba61c4c0873d391e987982fbcd3</sha1>
6
7 <!-- invalide -->
8 <sha1>test</sha1>
```

La restriction minLength

La restriction **minLength** permet de définir le nombre minimum de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2   <xsd:simpleType>
3     <xsd:restriction base="type_de_base">
4       <xsd:minLength value="ma_valeur" />
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:element>
```

La restriction maxLength

La restriction **maxLength** permet de définir le nombre maximum de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2   <xsd:simpleType>
3     <xsd:restriction base="type_de_base">
4       <xsd:maxLength value="ma_valeur" />
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:element>
```

La restriction enumeration

La restriction **enumeration**, comme son nom le laisse deviner, cette restriction permet d'énumérer la liste des valeurs possibles pour un élément ou un attribut.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2   <xsd:simpleType>
3     <xsd:restriction base="type_de_base">
4       <xsd:enumeration value="valeur1" />
5       <xsd:enumeration value="valeur2" />
6       <!-- liste des valeurs... -->
7     </xsd:restriction>
```

```

8 |     </xsd:simpleType>
9 | </xsd:element>
```

Afin d'illustrer cette restriction, prenons l'exemple d'une personne comme nous l'avons déjà fait à plusieurs reprises dans ce tutoriel. Une personne possède un nom, un prénom, et est normalement un homme ou une femme. Écrivons alors la règle d'un Schéma XML permettant de décrire une personne :

```

1 <xsd:complexType name="personne">
2   <xsd:sequence>
3     <xsd:element name="nom" type="xsd:string"/>
4     <xsd:element name="prenom" type="xsd:string"/>
5   </xsd:sequence>
6   <xsd:attribute name="sexe">
7     <xsd:simpleType>
8       <xsd:restriction base="xsd:string">
9         <xsd:enumeration value="masculin" />
10        <xsd:enumeration value="feminin" />
11       </xsd:restriction>
12     </xsd:simpleType>
13   </xsd:attribute>
14 </xsd:complexType>

1 <!-- valide -->
2 <personne sexe="masculin">
3   <nom>DUPONT</nom>
4   <prenom>Robert</prenom>
5 </personne>
6
7 <!-- valide -->
8 <personne sexe="feminin">
9   <nom>DUPONT</nom>
10  <prenom>Robert</prenom>
11 </personne>
12
13 <!-- invalide -->
14 <personne sexe="pomme">
15   <nom>DUPONT</nom>
16   <prenom>Robert</prenom>
17 </personne>
```

La restriction whiteSpace

La restriction **whiteSpace** permet de spécifier le comportement à adopter par un élément lorsqu'il contient des espaces. Les espaces peuvent être de différentes formes comme par exemple les tabulations, les retours à la ligne, etc.

3 valeurs sont possibles :

- **Preserve** : cette valeur permet de *préserver* tous les espaces.
- **Replace** : cette valeur permet de *remplacer* tous les espaces (tabulations, retour à la ligne, etc.) par des espaces « simples ».

- **Collapse** : cette valeur permet de *supprimer* les espaces en début et fin de chaîne, de remplacer les tabulations et les retours à la ligne par un espace « simple » et de remplacer les espaces multiples par un espace « simple ».

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2   <xsd:simpleType>
3     <xsd:restriction base="type_de_base">
4       <xsd:whiteSpace value="ma_valeur" />
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:element>
```

La restriction pattern

La restriction **pattern** permet de définir des expressions rationnelles (également appelées expressions régulières). Une expression rationnelle est un motif qui permet de décrire le contenu attendu. Ainsi à l'aide des expressions rationnelles, il est possible de dire que le contenu attendu doit forcément débuter par une majuscule, se terminer un point, débuter par un caractère spécial, ne pas contenir la lettre « z », etc.

Les **expressions rationnelles** sont un langage à part entière sur lequel nous n'allons pas revenir dans ce cours, mais ne vous inquiétez pas, vous trouverez facilement sur Internet de nombreuses ressources francophones sur le sujet.

Voici sa syntaxe :

```
1 <xsd:element name="mon_nom">
2   <xsd:simpleType>
3     <xsd:restriction base="type_de_base">
4       <xsd:pattern value="ma_valeur" />
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:element>
```

Afin d'illustrer cette dernière restriction, prenons l'exemple du prénom non composé d'une personne. On souhaite qu'il débute par une majuscule et soit suivi par plusieurs caractères minuscules :

```
1 <xsd:complexType name="prenom">
2   <xsd:simpleType>
3     <xsd:restriction base="xsd:string">
4       <xsd:pattern value="[A-Z][a-z]+" />
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:complexType>

1 <!-- valide -->
2 <prenom>Robert</prenom>
3
4 <!-- valide -->
5 <prenom>Zozor</prenom>
```

```

6 |
7 | <!-- invalide -->
8 | <prenom>bernard</prenom>
```

L'héritage par extension

Dans cette seconde partie, nous allons voir le second type d'héritage : l'**héritage par extension**.

Définition

Une **extension** est une notion qui permet d'ajouter des informations à un type existant. On peut, par exemple, vouloir ajouter un élément ou un attribut.

Lorsque l'on déclare une extension sur un élément, c'est toujours le mot clef « base » qui est utilisé :

```

1 | <!-- contenu complexe -->
2 | <xsd:complexType name="mon_nom">
3 |   <xsd:complexContent>
4 |     <xsd:extension base="type_de_base">
5 |       <!-- détail de l'extension -->
6 |     </xsd:restriction>
7 |   </xsd:complexContent>
8 | </xsd:complexType>
9 |
10 | <!-- contenu simple -->
11 | <xsd:complexType name="mon_nom">
12 |   <xsd:simpleContent>
13 |     <xsd:extension base="type_de_base">
14 |       <!-- détail de l'extension -->
15 |     </xsd:restriction>
16 |   </xsd:simpleContent>
17 | </xsd:complexType>
```

Exemple

Je vous propose de mettre en application cette nouvelle notion d'héritage par extension au travers d'un exemple. Reprenons les données clientes d'une banque que nous manipulions dans le chapitre précédent. Dans cette exemple, nous avions défini un type « *compte* » appliqué au *compte courant* et au *livret A* de notre client :

```

1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3 |   <!-- déclaration des éléments -->
4 |   <xsd:element name="montant" type="xsd:double" />
5 | 
```

```
6     <xsd:element name="identite" maxOccurs="unbounded" >
7         <xsd:complexType>
8             <xsd:sequence>
9                 <xsd:element name="nom" type="xsd:string" />
10                <xsd:element name="prenom" type="xsd:string" />
11            </xsd:sequence>
12        </xsd:complexType>
13    </xsd:element>
14
15    <xsd:complexType name="compte">
16        <xsd:sequence>
17            <xsd:element ref="montant" />
18        </xsd:sequence>
19    </xsd:complexType>
20
21    <xsd:element name="comptes">
22        <xsd:complexType>
23            <xsd:sequence>
24                <xsd:element name="livretA" type="compte" />
25                <xsd:element name="courant" type="compte" />
26            </xsd:sequence>
27        </xsd:complexType>
28    </xsd:element>
29
30    <xsd:element name="client">
31        <xsd:complexType>
32            <xsd:sequence>
33                <xsd:element ref="identite" />
34                <xsd:element ref="comptes" />
35            </xsd:sequence>
36        </xsd:complexType>
37    </xsd:element>
38
39    <!-- Schéma XML -->
40    <xsd:element name="banque">
41        <xsd:complexType >
42            <xsd:sequence>
43                <xsd:element ref="client" />
44            </xsd:sequence>
45        </xsd:complexType>
46    </xsd:element>
47 </xsd:schema>
```

Imaginons maintenant que le compte courant et le livret A soient un peu différents. Par exemple, un compte courant n'a généralement pas de taux d'intérêts tandis que le livret A en a un. Malgré ce petit changement, le livret A et le compte courant restent sensiblement identiques. C'est là que l'héritage par extension intervient. Nous allons étendre le type compte en y ajoutant un attribut pour créer ainsi un nouveau type : celui d'un compte avec des intérêts.

```

1  <!-- le montant -->
2  <xsd:element name="montant" type="xsd:double" />
3
4  <!-- compte sans intérêts -->
5  <xsd:complexType name="compte">
6      <xsd:sequence>
7          <xsd:element ref="montant" />
8      </xsd:sequence>
9  </xsd:complexType>
10
11 <!-- compte avec intérêts grâce à l'héritage par extension -->
12 <xsd:complexType name="compteInteret">
13     <xsd:complexContent>
14         <xsd:extension base="compte">
15             <xsd:attribute name="interet" type="xsd:float" />
16         </xsd:extension>
17     </xsd:complexContent>
18 </xsd:complexType>
```

Les identifiants

Nous avons déjà vu que dans un Schéma XML, il est possible d'identifier des ressources et d'y faire référence grâce aux mots clefs **ID** et **IDREF**.

Il est cependant possible d'aller plus loin et d'être encore plus précis grâce à 2 nouveaux mots clefs : **key** et **keyref**.

 Pour bien comprendre la suite de chapitre, il est nécessaire de connaître le fonctionnement d'une technologie qu'on appelle **XPath**, technologie que nous aborderons en plus en détail dans la prochaine partie. Pour le moment, retenez simplement que cette technologie nous permet de sélectionner avec précision des éléments formant un document XML.

La syntaxe

L'élément key

Au sein d'un Schéma XML, l'élément `<key />` est composé :

- D'un élément `<selector />` contenant une expression XPath afin d'indiquer l'élément à référencer.
- D'un ou plusieurs éléments `<field />` contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant.

Ce qui nous donne :

```
1 | <xsd:key name="nom_identifiant">
```

```
2 |     <selector xpath="expression_XPath" />
3 |     <!-- liste d'éléments field -->
4 |     <field xpath="expression_XPath" />
5 | </xsd:key>
```

L'élément keyref

L'élément `<keyref />` se construit sensiblement comme l'élément `<key />`. Il est donc composé :

- D'un élément `<selector />` contenant une expression XPath afin d'indiquer l'élément à référencer.
- D'un ou plusieurs éléments `<field />` contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant.

Ce qui nous donne :

```
1 | <xsd:keyref name="nom" refer="nom_identifiant">
2 |     <selector xpath="expression_XPath" />
3 |     <!-- liste d'éléments field -->
4 |     <field xpath="expression_XPath" />
5 | </xsd:keyref>
```

Exemple

Afin d'illustrer cette nouvelle notion, je vous propose de d'étudier le document XML suivant :

```
1 | <famille>
2 |     <pere id="PER-1" />
3 |     <enfant id="PER-2" pere="PER-1" />
4 | </famille>
```

Dans cet exemple, une famille est composée d'un père et d'un enfant dont chacun possède un identifiant unique au travers de l'attribut `id`. L'enfant possède également un attribut `pere` qui contient l'identifiant de son père.

Je vous propose de construire ensemble le Schéma XML correspond au document XML. Commençons par décrire l'élément `<pere />` :

```
1 | <xsd:element name="pere">
2 |     <xsd:complexType>
3 |         <xsd:attribut name="id" type="xsd:NCName" />
4 |     </xsd:complexType>
5 | </xsd:element>
```

Continuons avec l'élément `<enfant />` :

```
1 | <xsd:element name="enfant">
2 |     <xsd:complexType>
```

```

3      <xsd:attribut name="id" type="xsd:NCName" />
4      <xsd:attribut name="pere" type="xsd:NCName" />
5    </xsd:complexType>
6  </xsd:element>
```

Terminons avec l'élément `<famille />` :

```

1 <xsd:element name="famille">
2   <xsd:complexType>
3     <xsd:sequence>
4       <xsd:element ref="pere" />
5       <xsd:element ref="enfant" />
6     </xsd:sequence>
7   </xsd:complexType>
8 </xsd:element>
```

Modifions ce dernier élément afin d'y ajouter nos identifiants. Pour le moment, je vous demande d'accepter les expressions XPath présentes dans le Schéma XML. Dans la partie suivante, vous serez normalement en mesure de les comprendre.

```

1 <xsd:element name="famille">
2   <xsd:complexType>
3     <xsd:sequence>
4       <xsd:element ref="pere" />
5       <xsd:element ref="enfant" />
6     </xsd:sequence>
7   </xsd:complexType>
8
9   <!-- identifiant du père -->
10  <xsd:key name="pereId">
11    <xsd:selector xpath=".//pere" />
12    <xsd:field xpath="@id" />
13  </xsd:key>
14
15  <!-- identifiant de l'enfant -->
16  <xsd:key name="enfantId">
17    <xsd:selector xpath=".//enfant" />
18    <xsd:field xpath="@id" />
19  </xsd:key>
20
21  <!-- référence à l'identifiant du père dans l'élément
22    enfant -->
23  <xsd:key name="pereIdRef" refer="pereId">
24    <xsd:selector xpath=".//enfant" />
25    <xsd:field xpath="@pere" />
26  </xsd:key>
27 </xsd:element>
```

Un exemple avec EditiX

Pour conclure ce chapitre, je vous propose de voir ensemble **comment écrire un Schéma XML avec EditiX**.

Pour faire simple, reprenons l'exemple de notre banque vu dans le chapitre sur la réutilisation des types.

Création du document XML

La **création du document XML** n'a rien de bien compliqué, nous l'avons déjà vu ensemble dans la partie précédente.

Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil à la page 21.

Voici le document que vous devez écrire :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <banque xsi:noNamespaceSchemaLocation="banque.xsd" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance">
3   <!-- 1er client de la banque -->
4   <client>
5     <!-- identité du client -->
6     <identite>
7       <nom>NORRIS</nom>
8       <prenom>Chuck</prenom>
9     </identite>
10
11    <!-- liste des comptes bancaires du client -->
12    <comptes>
13      <livretA>
14        <montant>2500</montant>
15      </livretA>
16      <courant>
17        <montant>4000</montant>
18      </courant>
19    </comptes>
20  </client>
21 </banque>
```

Si vous essayez de lancer la vérification du document, vous devriez normalement avoir le message d'erreur suivant (voir la figure 14.1).



FIGURE 14.1 – Message d'erreur indiquant que le Schéma XML est introuvable

Ce message est pour le moment complètement normal puisque nous n'avons pas encore créé notre document XSD.

Création du document XSD

Pour créer un nouveau document, vous pouvez sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier **[Ctrl] + [N]**.

Dans la liste qui s'affiche, sélectionnez **W3C XML Schema**, ainsi qu'indiqué sur la figure 14.2.

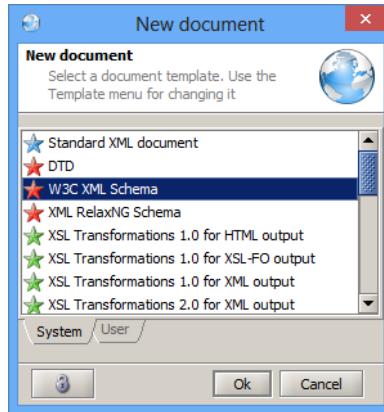


FIGURE 14.2 – Crédit d'un Schéma XML

Votre document XSD n'est normalement pas vierge. Voici ce que vous devriez avoir :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4
5 </xsd:schema>
```

Replacez le contenu par notre véritable Schéma XML :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <!-- déclaration des éléments -->
4   <xsd:element name="montant" type="xsd:double" />
5
6   <xsd:element name="identite" >
7     <xsd:complexType>
8       <xsd:sequence>
9         <xsd:element name="nom" type="xsd:string" />
10        <xsd:element name="prenom" type="xsd:string" />
11      </xsd:sequence>
12    </xsd:complexType>
13  </xsd:element>
14
15  <xsd:complexType name="compte">
16    <xsd:sequence>
```

```
17         <xsd:element ref="montant" />
18     </xsd:sequence>
19 </xsd:complexType>
20
21 <xsd:element name="comptes">
22     <xsd:complexType>
23         <xsd:sequence>
24             <xsd:element name="livretA" type="compte" />
25             <xsd:element name="courant" type="compte" />
26         </xsd:sequence>
27     </xsd:complexType>
28 </xsd:element>
29
30 <xsd:element name="client">
31     <xsd:complexType>
32         <xsd:sequence>
33             <xsd:element ref="identite" />
34             <xsd:element ref="comptes" />
35         </xsd:sequence>
36     </xsd:complexType>
37 </xsd:element>
38
39 <!-- Schéma XML -->
40 <xsd:element name="banque">
41     <xsd:complexType>
42         <xsd:sequence>
43             <xsd:element ref="client" maxOccurs="unbounded"
44                 />
45         </xsd:sequence>
46     </xsd:complexType>
47 </xsd:element>
48 </xsd:schema>
```

Enregistrez ensuite votre document avec le nom **banque.xsd** au même endroit que votre document XML.

Vérification du Schéma XML

Vous pouvez vérifier que votre Schéma XML n'a pas d'erreur de syntaxe en sélectionnant dans la barre de menu **XML** puis **Check this document** ou encore en utilisant le raccourci clavier **Ctrl** + **K**. Vous devriez normalement avoir le message suivant (voir figure 23.2).

Vérification du document XML

Il est maintenant temps de vérifier que le document XML est **valide**!

Pour ce faire, cliquez sur l'icône adéquate ou sélectionnez dans la barre de menu **XML**



FIGURE 14.3 – Message indiquant que le Schéma XML ne contient aucune erreur de syntaxe

puis **Check this document** ou utilisez le raccourci clavier **[Ctrl] + [K]**.

Un message doit normalement s'afficher (voir la figure 23.2).



FIGURE 14.4 – Message indiquant que le document XML est valide

En résumé

- Le nombre d'occurrences d'un élément s'exprime grâce aux mots clefs **minOccurs** et **maxOccurs**.
- Le mot clef **ref** permet de faire référence à des éléments dans le but de les réutiliser plusieurs fois au sein du Schéma XML.
- L'**héritage** permet de réutiliser des éléments d'un Schéma XML pour en construire de nouveaux.
- Il existe 2 types d'héritages : l'**héritage par restriction** et l'**héritage par extension**.

Chapitre 15

TP : Schéma XML d'un répertoire

Difficulté : 

Souvenez vous, lorsque nous avons étudié les **DTD**, nous avions mis en pratique notre apprentissage à travers la définition DTD d'un **répertoire**.

Pour ce nouveau TP, le sujet ne change pas, il convient toujours d'écrire la définition d'un répertoire. Ce qui change c'est la technologie à utiliser. Cette fois-ci, c'est un Schéma XML que je vous demande d'écrire !



L'énoncé

Le but de ce TP est de créer le Schéma XML du répertoire que nous avons déjà vu. Pour rappel, voici les informations que l'on souhaite connaître pour chaque personne :

- Son sexe (homme ou femme).
- Son nom.
- Son prénom.
- Son adresse.
- Un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.).
- Une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici le document XML que nous avions construit :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <repertoire>
4     <!-- John DOE -->
5     <personne sexe="masculin">
6         <nom>DOE</nom>
7         <prenom>John</prenom>
8         <adresse>
9             <numero>7</numero>
10            <voie type="impasse">impasse du chemin</voie>
11            <codePostal>75015</codePostal>
12            <ville>PARIS</ville>
13            <pays>FRANCE</pays>
14        </adresse>
15        <telephones>
16            <telephone type="fixe">01 02 03 04 05</telephone>
17            <telephone type="portable">06 07 08 09 10</
18                telephone>
19        </telephones>
20        <emails>
21            <email type="personnel">john.doe@wanadoo.fr</email>
22            <email type="professionnel">john.doe@societe.com</
23                email>
24        </emails>
25    </personne>
26
27    <!-- Marie POPPINS -->
28    <personne sexe="feminin">
29        <nom>POPPINS</nom>
30        <prenom>Marie</prenom>
31        <adresse>
32            <numero>28</numero>
33            <voie type="avenue">avenue de la république</voie>
34            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
```

```
35      </adresse>
36      <telephones>
37          <telephone type="bureau">04 05 06 07 08</telephone>
38      </telephones>
39      <emails>
40          <email type="professionnel">contact@poppins.fr</
41              email>
42      </emails>
43  </personne>
</repertoire>
```

Une solution

Comme à chaque fois, je vous fais part de ma solution. Mais étant donné qu'elle est un peu longue, je vous invite à utiliser le code web suivant pour y accéder.

▷ [Voir ma solution](#)
[Code web : 776572](#)

Un bref commentaire

Dans cette solution, je suis allé au plus simple. Libre à vous de créer de nouvelles règles si vous souhaitez par exemple utiliser un pattern précis pour les numéros de téléphone ou les adresses e-mails.

Troisième partie

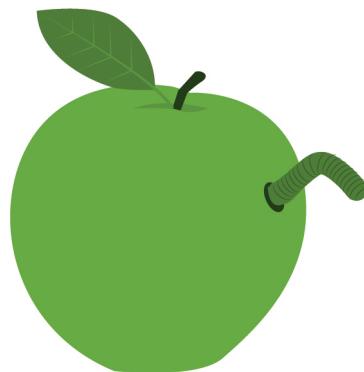
Traitez vos données XML

DOM : Introduction à l'API

Difficulté : 

Dans la seconde partie, nous avons eu l'occasion d'étudier 2 technologies permettant d'écrire la définition d'un document XML. Cette troisième partie est également l'occasion de découvrir 2 nouvelles technologies qu'il est possible d'utiliser en parallèle d'un document XML : l'API DOM et XPath.

Ces 2 technologies vont nous permettre d'extraire et exploiter les informations contenues dans un document XML. Débutons immédiatement la première des deux : l'API DOM.



Qu'est-ce que L'API DOM ?

La petite histoire de DOM

DOM ou **Document Object Model**, son nom complet, est ce qu'on appelle un **parseur XML**, c'est-à-dire, une technologie grâce à laquelle il est possible de lire un document XML et d'en extraire différentes informations (éléments, attributs, commentaires, etc...) afin de les exploiter.

Comme pour la plupart des technologies abordées dans ce tutoriel, DOM est un standard du W3C et ce, depuis sa première version en 1998. Au moment où j'écris ces lignes, la technologie en est à sa troisième version.

Il est très important de noter que DOM est une recommandation complètement *indépendante* de toute plate-forme et langage de programmation. Au travers de DOM, le W3C fournit une recommandation, c'est-à-dire une manière d'exploiter les documents XML.

Aujourd'hui, la plupart des langages de programmation propose leur implémentation de DOM :

- C.
- C ++.
- Java.
- C#.
- Perl.
- PHP.
- etc.



Dans les chapitres suivants, les exemples seront illustrés à l'aide du langage Java.

L'arbre XML

Dans le chapitre précédent, je vous disais que DOM est une technologie complètement indépendante de toute plate-forme et langage de programmation et qu'elle se contente de fournir une manière d'exploiter les documents XML.

En réalité, lorsque votre document XML est lu par un parseur DOM, le document est représenté en mémoire sous la forme d'un arbre dans lequel les différents éléments sont liés les uns aux autres par une relation parent/enfant. Il est ensuite possible de passer d'un élément à un autre via un certain nombre de fonctions que nous verrons dans le chapitre suivant.

Je vous propose d'illustrer cette notion d'arbre grâce à un exemple. Soit le document XML suivant :

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</
10      telephone>
11   </personne>
12 </repertoire>

```

Voici à la figure 18.3 ce à quoi ressemble l'arbre une fois modélisé.

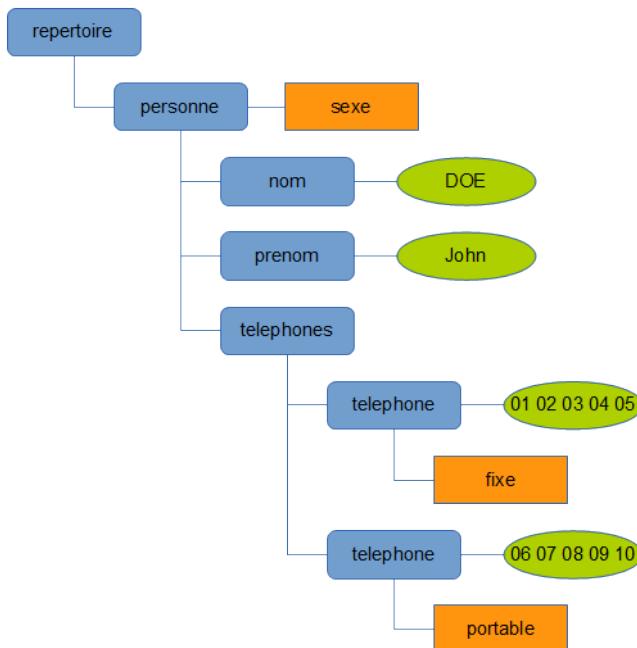


FIGURE 16.1 – Modélisation d'un arbre XML

Pour bien comprendre à quoi correspondent les couleurs et les formes, voici la légende en figure 18.2.

Dans nos futurs programmes c'est donc ce genre d'arbres que nous allons parcourir afin d'obtenir les informations que l'on souhaite exploiter en passant d'un élément à un autre. Mais avant de voir comment procéder, je vous propose de revenir sur le vocabulaire utilisé par DOM.

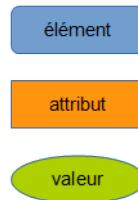


FIGURE 16.2 – Légende des icônes utilisées dans l'arbre XML

Le vocabulaire et les principaux éléments

Dans ce chapitre, nous allons découvrir ensemble le vocabulaire utilisé par le **parseur DOM**. Nous allons également en profiter pour faire le tour des principaux éléments en terme de programmation.



Lors de la description des différents éléments, je vais tenter d'être le plus neutre possible en me détachant de tout langage de programmation. Cependant, des connaissances en programmation objet sont nécessaires pour comprendre ce chapitre.

Document

Définition

Le **document** comme son nom le laisse deviner désigne le document XML dans son ensemble. Il est donc composé :

- Du **prologue**.
- Du **corps**.

La classe

Grâce à la classe **Document** nous allons pouvoir exploiter aussi bien le **prologue** que le **corps** de nos documents XML. Cette classe va également se révéler indispensable lorsque nous allons vouloir créer ou modifier des documents XML. En effet, via les nombreuses méthodes proposées, nous allons pouvoir ajouter des éléments, des commentaires, des attributs, etc.

Node

Définition

Un **Node** ou **Nœud** en français peut-être véritablement considéré comme l'élément de base d'un arbre XML. Ainsi, toute branche ou feuille est un nœud. Un élément est donc un nœud, tout comme une valeur et un attribut.

Je vous propose de reprendre le schéma vu précédemment et de tenter d'identifier 3 nœuds parmi tous ceux présents (voir la figure 16.3).

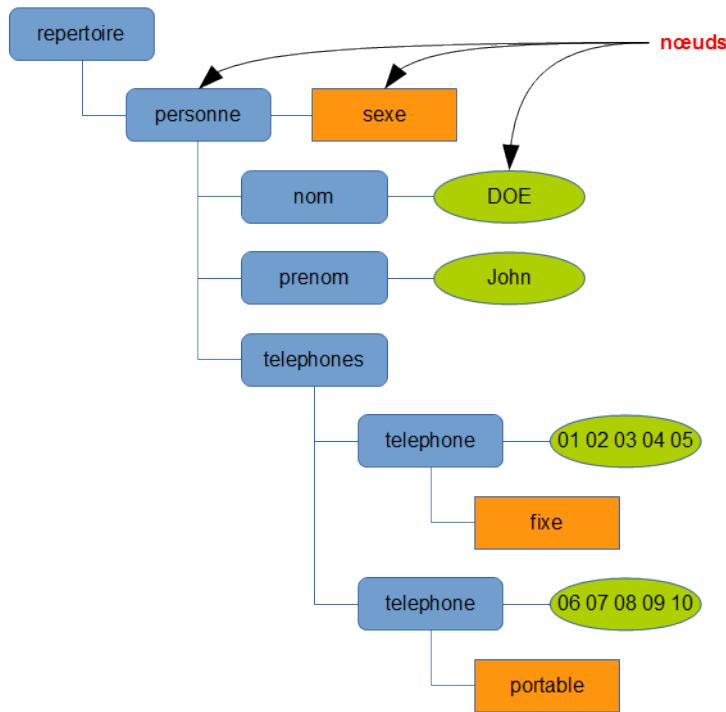


FIGURE 16.3 – Des nœuds

La classe

La classe **Node** nous permet d'obtenir un certain nombre d'informations lors de l'exploitation d'un document XML. Ainsi, il est possible d'obtenir le type du nœud (attribut, valeur, etc.) son nom, sa valeur, la liste des nœuds fils, le nœud parent, etc. Cette classe propose également un certain nombre de méthodes qui vont nous aider à créer et modifier un document XML en offrant par exemple la possibilité de supprimer un nœud, ou d'en remplacer un par un autre, etc.

Element

Définition

Un **Element** représente une balise d'un document XML. Si l'on reprend le schéma de l'arbre XML du dessus, les éléments sont en bleu.

La classe

La classe **Element**, en plus de nous fournir le nom de la balise, nous offre de nombreuses fonctionnalités comme par exemple la possibilité de récupérer les informations d'un attribut ou encore de récupérer la liste des noeuds d'un élément portant un nom spécifique.

Attr

Définition

Un **Attr** désigne un attribut. Si l'on reprend le schéma de l'arbre XML du dessus, les attributs sont en orange.

La classe

La classe **Attr**, permet d'obtenir un certain nombre d'information concernant les attributs comme son nom ou encore sa valeur. Cette classe va également nous être utile lorsque l'on voudra créer ou modifier des documents XML.

Text

Définition

Un **Text** désigne le contenu d'une balise. Si l'on reprend le schéma de l'arbre XML du dessus, ils sont en vert.

La classe

En plus de la donnée textuelle, la classe **Text**, permet de facilement modifier un document XML en proposant par exemple des méthodes de suppression ou de remplacement de contenu.

Les autres éléments

Il est presque impossible de présenter tous les éléments du DOM vu la densité de la technologie. Sachez cependant que nous avons vu les principaux et qu'un exemple

d'utilisation de l'implémentation Java est prévu dans le chapitre suivant.

Concernant les éléments non décrits, il existe par exemple la classe **Comment** permettant de gérer les commentaires ou encore la classe **CDataSection** permettant de d'exploiter les sections **CData** d'un document XML.

Finalement, sachez que grâce à DOM, il est également possible de vérifier la validité d'un document XML à une définition DTD ou un Schéma XML.

En résumé

- DOM est une recommandation complètement indépendante de toute plate-forme et langage de programmation ;
- DOM exploite l'arbre XML d'un document XML.

Chapitre 17

DOM : Exemple d'utilisation en Java

Difficulté : 

Dans le chapitre précédent, nous venons donc de faire connaissance avec l'**API DOM**. Dans ce nouveau chapitre, nous allons manipuler l'**implémentation Java** de cette technologie en découvrant ensemble, étape par étape, comment lire et créer un document XML.

Pour ceux d'entre vous qui n'utilisent pas Java ou qui sont plus à l'aise avec un autre langage de programmation, n'hésitez pas à aller faire une petite visite sur le forum du site du Zéro pour trouver des réponses à toutes vos questions !



Lire un document XML

Le document XML

Avant de plonger dans le code, voici le document XML que nous allons tenter d'exploiter :

```
1 | <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 | <repertoire>
3 |     <!-- John DOE -->
4 |     <personne sexe="masculin">
5 |         <nom>DOE</nom>
6 |         <prenom>John</prenom>
7 |         <telephones>
8 |             <telephone type="fixe">01 02 03 04 05</telephone>
9 |             <telephone type="portable">06 07 08 09 10</
10 |                 telephone>
11 |             </telephones>
12 |         </personne>
13 |     </repertoire>
```

Mise en place du code

Étape 1 : récupération d'une instance de la classe « DocumentBuilderFactory »

Avant même de pouvoir prétendre créer un **parseur DOM**, nous devons récupérer une instance de la classe `DocumentBuilderFactory`. C'est à partir de cette instance que dans l'étape suivante nous pourrons créer notre parseur.

Pour récupérer une instance de la classe « `DocumentBuilderFactory` » une ligne de code suffit :

```
1 | final DocumentBuilderFactory factory = DocumentBuilderFactory.
|     newInstance();
```

Cette ligne de code s'accompagne de l'importation du package :

```
1 | import javax.xml.parsers.DocumentBuilderFactory;
```

Étape 2 : création d'un parseur

La seconde étape consiste à créer un parseur à partir de notre variable `factory` créée dans l'étape 1.

Une nouvelle fois, une seule ligne de code est suffisante :

```
1 | final DocumentBuilder builder = factory.newDocumentBuilder();
```

Pour pouvoir utiliser la classe DocumentBuilder, le package suivant est nécessaire :

```
1 | import javax.xml.parsers.DocumentBuilder;
```

Puisque l'appel à la fonction newDocumentBuilder(), peut lever une exception, il convient de le placer dans un bloc de type **try/catch** :

```
1 | try {
2 |     final DocumentBuilder builder = factory.newDocumentBuilder
3 |         ();
4 | }
5 | catch (final ParserConfigurationException e) {
6 |     e.printStackTrace();
6 | }
```

La gestion de cette exception oblige à également importer le package :

```
1 | import javax.xml.parsers.ParserConfigurationException;
```

Étape 3 : création d'un Document

La troisième étape consiste à créer un **Document** à partir parseur de l'étape 2. Plusieurs possibilités s'offre alors à vous :

- A partir d'un fichier.
- A partir d'un flux.

Ce flux peut par exemple être le résultat de l'appel à un web service. Dans notre exemple, c'est un fichier qui est utilisé :

```
1 | final Document document= builder.parse(new File("repertoire.xml
2 | "));
```

Comme à chaque nouvelle instruction, des packages doivent être importés :

```
1 | import java.io.File;
2 | import org.w3c.dom.Document;
```

De nouvelles exceptions pouvant être levées, il convient également de modifier leurs captures dans des blocs **catch**. Voici alors ce que vous devriez avoir :

```
1 | try {
2 |     final DocumentBuilder builder = factory.newDocumentBuilder
3 |         ();
4 |     final Document document= builder.parse(new File("repertoire
5 |         .xml"));
6 |
7 | }
8 | catch (final ParserConfigurationException e) {
9 |     e.printStackTrace();
9 | }
10 | catch (final SAXException e) {
11 |     e.printStackTrace();
11 | }
```

```
10    }
11    catch (final IOException e) {
12        e.printStackTrace();
13    }
```

La gestion de ces nouvelles exceptions nous oblige également à importer quelques packages :

```
1 import java.io.IOException;
2 import org.xml.sax.SAXException;
```

Comme je vous le disais dans le chapitre précédent, un **Document** représente le document XML dans son intégralité. Il contient son **prologue** et son **corps**. Nous allons pouvoir le vérifier en affichant les éléments du prologue, à savoir :

- La version XML utilisée.
- L'encodage utilisé.
- S'il s'agit d'un document « standalone » ou non.

```
1 //Affiche la version de XML
2 System.out.println(document.getXmlVersion());
3
4 //Affiche l'encodage
5 System.out.println(document.getXmlEncoding());
6
7 //Affiche s'il s'agit d'un document standalone
8 System.out.println(document.getXmlStandalone());
```

A l'exécution du programme, voici ce que vous devriez avoir à l'écran :

```
UTF-8
1.0
true
```

Si l'on compare cet affichage au prologue du document XML, on se rend compte que les informations correspondent bien.

Étape 4 : récupération de l'Element racine

Dans cette quatrième étape, nous allons laisser de côté le prologue et tenter de nous attaquer au **corps** du document XML. Pour ce faire, nous allons extraire l'**Element racine** du Document. C'est à partir de lui que nous pourrons ensuite naviguer dans le reste du document.

Pour récupérer l'élément racine, il suffit d'écrire de faire appel à la fonction `getDocumentElement()` de notre document :

```
1 | final Element racine = document.getDocumentElement();
```

Une nouvelle fois, pour pouvoir utiliser la classe `Element`, le package suivant doit être importé :

```
1 | import org.w3c.dom.Element;
```

Nous pouvons dès maintenant vérifier que ce que nous venons de récupérer est bien l'élément racine de notre document en affichant son nom :

```
1 | System.out.println(racine.getNodeName());
```

Après exécution du programme, voici ce que vous devriez avoir à l'écran :

repertoire

Étape 5 : récupération des personnes

C'est à partir de cette étape que les choses sérieuses commencent ! En effet, dans cette cinquième étape, nous allons réellement parcourir le corps de notre document XML.

Il est possible de parcourir un document XML sans connaître sa structure. Il est donc possible de créer un code assez générique notamment grâce à la récursivité. Cependant, ce n'est pas la méthode que j'ai choisi d'utiliser dans ce chapitre. Nous allons donc parcourir notre document en partant du principe que nous connaissons sa structure.

Pour récupérer tous les noeuds enfants de la racine, voici la ligne de code à écrire :

```
1 | final NodeList racineNoeuds = racine.getChildNodes();
```

Il vous faudra également importer le package suivant :

```
1 | import org.w3c.dom.NodeList;
```

Nous pouvons également nous amuser à afficher le nom de chacun des noeuds via le code suivant :

```
1 | final int nbRacineNoeuds = racineNoeuds.getLength();
2 |
3 | for (int i = 0; i < nbRacineNoeuds; i++) {
4 |     System.out.println(racineNoeuds.item(i).getNodeName());
5 | }
```

Vous devriez alors avoir le résultat suivant :

#text
 #comment
 #text
 personne
 #text

On retrouve bien notre balise `<personne />` au milieu d'autres noeuds de type **text** et **comment**. Nous allons maintenant légèrement modifier notre boucle afin de n'afficher à l'écran que les noeuds étant des éléments. Grâce à la méthode `getNodeType()` de la classe **Node** :

```
1 for (int i = 0; i<nbRacineNoeuds; i++) {
2     if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE)
3         {
4             final Node personne = racineNoeuds.item(i);
5             System.out.println(personne.getNodeName());
6         }
6 }
```

A l'exécution de votre programme, vous devriez normalement avoir le résultat suivant :

```
personne
```

A noter : si vous avions voulu récupérer le commentaire, nous aurions comparé le type de notre noeud à la constante `Node.COMMENT_NODE`.

Avant de passer à l'affichage de la suite du document XML, nous allons tenter d'afficher le sexe de la personne, qui pour rappel est un attribut. Il existe plusieurs manières de faire plus ou moins génériques. La méthode la plus générique est de faire appelle à la méthode `getAttributes()` de la classe **Node** qui nous renvoie l'ensemble des attributs du noeud. Dans notre cas, nous allons utiliser la méthode `getAttribute(nom)` qui nous renvoie la valeur de l'attribut spécifié en paramètre. Cette méthode n'est cependant pas accessible à partir de la classe **Node**, il convient donc de « caster » notre noeud en un **Element** pour pouvoir l'appeler :

```
1 for (int i = 0; i<nbRacineNoeuds; i++) {
2     if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE)
3         {
4             final Element personne = (Element) racineNoeuds.item(i)
5                 ;
6             System.out.println(personne.getNodeName());
5             System.out.println("sexe : " + personne.getAttribute(
6                     "sexe"));
6         }
7 }
```

A l'écran devrait alors s'afficher le résultat suivant :

```
personne
sexe : masculin
```

Étape 6 : récupération du nom et du prénom

Nous allons maintenant récupérer le nom et le prénom des personnes présentes dans notre document XML. Puisque nous savons exactement ce que l'on souhaite récupérer, nous allons y accéder directement via la méthode `getElementsByTagName(name)` de l'objet **Element**. Cette méthode nous renvoie tous éléments contenus dans l'élément et portant le nom spécifié.

Pour mieux comprendre, voyons un exemple :

```
1 | final NodeList noms = personne.getElementsByTagName("nom");
```

Ainsi, nous venons de récupérer tous les éléments d'une personne ayant pour nom « nom ». Dans notre cas, nous savons qu'une personne ne peut avoir qu'un seul prénom, nous pouvons donc préciser que nous voulons le premier élément de la liste :

```
1 | final Element nom = (Element) personne.getElementsByTagName("nom").item(0);
```

Finalement, si l'on souhaite afficher le **Text** de la balise, il nous suffit d'appeler la méthode `getTextContent()` :

```
1 | System.out.println(nom.getTextContent());
```

Vous devriez alors voir s'afficher à l'écran le nom de la seule personne déclarée dans notre document XML :

```
DOE
```

Pour extraire le prénom d'une personne, la logique est exactement la même.

Étape 7 : récupération des numéros de téléphone

La septième et dernière étape de la lecture de notre document XML consiste à récupérer les numéros de téléphone d'une personne. La logique est sensiblement la même que dans l'étape 1 si ce n'est que le résultat de la méthode `getElementsByTagName(name)` nous renverra éventuellement plusieurs résultats. Il suffit alors de boucler sur les résultats pour afficher les valeurs et les attributs.

Voici le code qui devrait être écrit :

```
1 | final NodeList telephones = personne.getElementsByTagName("telephone");
2 | final int nbTelephonesElements = telephones.getLength();
3 |
4 | for(int j = 0; j<nbTelephonesElements; j++) {
5 |     final Element telephone = (Element) telephones.item(j);
6 |     System.out.println(telephone.getAttribute("type") + " : " +
7 |                         telephone.getTextContent());
```

Vous devriez alors voir s'afficher à l'écran la liste des numéros de téléphones :

```
fixe : 01 02 03 04 05
portable : 06 07 08 09 10
```

Nous venons donc de lire ensemble notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire en suivant le code web suivant.

▷ [Voir le code](#)
[Code web : 758642](#)

Ecrire un document XML

Le document XML

Dans le chapitre précédent, nous avons vu comment lire un document XML. Dans ce chapitre, je vous propose d'en créer un de toute pièce. Voici le document que nous allons créer :

```
1 | <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 | <repertoire>
3 |   <!-- John DOE -->
4 |   <personne sexe="masculin">
5 |     <nom>DOE</nom>
6 |     <prenom>John</prenom>
7 |     <telephones>
8 |       <telephone type="fixe">01 02 03 04 05</telephone>
9 |       <telephone type="portable">06 07 08 09 10</
10 |      telephone>
11 |    </telephones>
12 |  </personne>
13 | </repertoire>
```

Je suis sûr que vous le connaissez.

Mise en place du code

Étape 1 : récupération d'une instance de la classe « DocumentBuilderFactory »

Comme pour la lecture d'un document XML, la première étape consiste à récupérer une instance de la classe `DocumentBuilderFactory`. C'est à partir de cette instance que notre parseur sera créé dans l'étape suivante :

```
1 | final DocumentBuilderFactory factory = DocumentBuilderFactory.
|   newInstance();
```

N'oubliez pas d'importer le package :

```
1 | import javax.xml.parsers.DocumentBuilderFactory;
```

Étape 2 : création d'un parseur

La seconde étape est également commune à la lecture d'un document XML. Ainsi, nous allons créer un parseur à partir de notre variable `factory` créée dans l'étape précédente :

```
1 | final DocumentBuilder builder = factory.newDocumentBuilder();
```

Cette ligne de code s'accompagne de l'importation du package suivant :

```
1 | import javax.xml.parsers.DocumentBuilder;
```

Bien que nous l'ayons déjà vu dans le chapitre précédent, n'oubliez pas qu'une exception peut être levée, c'est pourquoi cette instruction doit être placée dans un bloc de type **try/catch** :

```
1 | try {
2 |     final DocumentBuilder builder = factory.newDocumentBuilder
3 |     ();
4 | } catch (final ParserConfigurationException e) {
5 |     e.printStackTrace();
6 | }
```

La gestion de cette exception nous oblige à également importer le package suivant :

```
1 | import javax.xml.parsers.ParserConfigurationException;
```

Étape 3 : création d'un Document

La troisième étape consiste à créer un **Document** vierge. Ce document est créé à partir de notre parseur :

```
1 | final Document document= builder.newDocument();
```

Pour pouvoir utiliser la classe **Document**, n'oubliez pas d'importer le package suivant :

```
1 | import org.w3c.dom.Document;
```

Étape 4 : création de l'Element racine

Dans cette quatrième étape, nous allons créer l'élément racine de notre document XML, à savoir la balise `<repertoire />`. La création de l'élément racine se fait via notre parseur et plus particulièrement la fonction `createElement()` qui prend en paramètre le nom que l'on souhaite donner à la balise :

```
1 | final Element racine = document.createElement("repertoire");
```

A noter que l'utilisation de la classe **Element** s'accompagne de l'importation du package :

```
1 | import org.w3c.dom.Element;
```

Maintenant que notre élément racine est déclaré, nous pouvons l'ajouter à notre document :

```
1 | document.appendChild(racine);
```

Étape 5 : création d'une personne

Si l'on regarde le document XML que l'on doit créer, on s'aperçoit qu'avant de créer la balise <personne/>, nous devons créer un commentaire.

La création d'un commentaire n'est pas plus compliquée que la création d'une balise et se fait via la fonction `createComment()` du parseur qui prend comme paramètre le fameux commentaire :

```
1 | final Comment commentaire = document.createComment("John DOE");
```

Pour pouvoir déclarer un commentaire, n'oubliez pas d'importer le package suivant :

```
1 | import org.w3c.dom.Comment;
```

Il convient ensuite d'ajouter notre commentaire à la suite de notre document et plus spécifiquement à la suite de notre élément racine. Si l'on se réfère à l'arbre XML, le commentaire est réellement un sous élément de l'élément racine. C'est pourquoi celui-ci est ajouté en tant qu'enfant de l'élément racine :

```
1 | racine.appendChild(commentaire);
```

Nous pouvons maintenant nous attaquer à la création de la balise <personne />. Il s'agit d'un élément au même titre que l'élément racine que nous avons déjà vu. Puisque la balise est au même niveau que le commentaire, il convient de l'ajouter en tant qu'enfant de l'élément racine :

```
1 | final Element personne = document.createElement("personne");
2 | racine.appendChild(personne);
```

Si l'on s'arrête ici, on omet d'ajouter l'attribut « sexe » pourtant présent dans la balise <personne /> du document XML que l'on souhaite créer. Ajouter un attribut à un élément est en réalité très simple et se fait via la méthode `setAttribute()` de la classe `Element`. Cette méthode prend 2 paramètres : le nom de l'attribut et sa valeur.

```
1 | personne.setAttribute("sexe", "masculin");
```

Étape 6 : création du nom et du prénom

En soit, la création des balises <nom /> et <prenom /> n'a rien de compliqué. En effet, nous avons déjà créé ensemble plusieurs éléments.

```
1 | final Element nom = document.createElement("nom");
2 | final Element prenom = document.createElement("prenom");
3 |
4 | personne.appendChild(nom);
5 | personne.appendChild(prenom);
```

Ici, La nouveauté concerne le renseignement de la valeur contenu dans les balises, à savoir John DOE dans notre exemple. Pour ce faire, il convient d'ajouter à nos balises un enfant de type `Text`. Cet enfant doit être créé avec la méthode `createTextNode()` du document qui prend en paramètre la valeur :

```
1 | nom.appendChild(document.createTextNode("DOE"));
2 | prenom.appendChild(document.createTextNode("John"));
```

Étape 7 : création des numéros de téléphone

Je vais aller très vite sur cette étape en vous fournissant directement le code source. En effet, cette septième étape ne contient rien de nouveau par rapport à ce que nous avons vu jusqu'ici :

```
1 | final Element telephones = document.createElement("telephones")
2 | ;
3 |
4 | final Element fixe = document.createElement("telephone");
5 | fixe.appendChild(document.createTextNode("01 02 03 04 05"));
6 | fixe.setAttribute("type", "fixe");
7 |
8 | final Element portable = document.createElement("telephone");
9 | portable.appendChild(document.createTextNode("06 07 08 09 10"));
10 |
11 | portable.setAttribute("type", "portable");
12 |
13 | telephones.appendChild(fixe);
14 | telephones.appendChild(portable);
15 | personne.appendChild(telephones);
```

Étape 8 : affichage du résultat

Il est maintenant temps de passer à la dernière étape qui consiste à afficher notre document XML fraîchement créé. Deux possibilités s'offrent à nous :

- Dans un document XML.
- Dans la console de l'IDE.

Ne vous inquiétez pas, les 2 possibilités seront abordées dans ce tutoriel.

Pour pouvoir afficher notre document XML, nous allons avoir besoin de plusieurs objets Java. Le premier est une instance de la classe TransformerFactory :

```
1 | final TransformerFactory transformerFactory =
2 |     TransformerFactory.newInstance();
```

La récupération de cette instance s'accompagne de l'importation du package suivant :

```
1 | import javax.xml.transform.TransformerFactory;
```

Nous allons utiliser cette instance pour créer un objet Transformer. C'est grâce à lui que nous pourrons afficher notre document XML par la suite :

```
1 | final Transformer transformer = transformerFactory.
2 |     newTransformer();
```

A noter que la fonction newTransformer() peut lever une exception de type TransformerConfigurationException qu'il est important de capturer via un bloc catch.

N'oubliez pas d'importer les packages suivants :

```
1 | import javax.xml.transform.Transformer;
2 | import javax.xml.transform.TransformerConfigurationException;
```

Pour afficher le document XML, nous utiliserons la méthode transform() de notre transformer. Cette méthode prend en compte 2 paramètres :

- La source.
- La sortie.

```
1 | transformer.transform(source, sortie);
```

A noter qu'une exception de type TransformerConfigurationException est susceptible d'être levée :

```
1 | import javax.xml.transform.TransformerException;
```

En ce qui nous concerne, la source que l'on souhaite afficher est notre document XML. Cependant, nous ne pouvons pas passer notre objet document tel quel. Il convient de le transformer légèrement sous la forme d'un objet DOMSource :

```
1 | final DOMSource source = new DOMSource(document);
```

Pour pouvoir utiliser cette classe, il convient d'importer le package suivant :

```
1 | import javax.xml.transform.dom.DOMSource;
```

Maintenant que nous avons la source, occupons nous de la sortie. La sortie est en réalité un objet StreamResult. C'est ici que nous allons préciser si nous souhaitons afficher notre document dans un fichier ou dans la console de notre IDE :

```
1 | //Code à utiliser pour afficher dans un fichier
2 | final StreamResult sortie = new StreamResult(new File("F:\\\\file
   .xml"));
3 |
4 | //Code à utiliser pour afficher dans la console
5 | final StreamResult sortie = new StreamResult(System.out);
```

Encore une fois, l'importation d'un package est nécessaire :

```
1 | import javax.xml.transform.stream.StreamResult;
```

Avant d'exécuter notre programme, il nous reste encore quelques petits détails à régler : l'écriture du prologue et le formatage de l'affichage.

Commençons par le prologue. Nous allons renseigner ses différentes propriétés via la méthode setOutputProperty() de notre transformer qui prend en paramètre le nom du paramètre et sa valeur :

```
1 | transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
2 | transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
3 | transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
```

Pour pouvoir utiliser les constantes de la classe OutputKeys, il convient d'importer le package suivant :

```
1 | import javax.xml.transform.OutputKeys;
```

Si nous exécutons notre programme maintenant, tout sera écrit sur une seule ligne, ce qui n'est pas très lisible, vous en conviendrez. C'est pourquoi nous allons donner quelques règles de formatage à notre transformeur. En effet, ce que l'on souhaite c'est que notre document soit indenté. Chaque niveau différent de notre document XML sera alors décalé de 2 espaces :

```
1 | transformer.setOutputProperty(OutputKeys.INDENT, "yes");
2 | transformer.setOutputProperty("{http://xml.apache.org/xslt}
    indent-amount", "2");
```

Nous venons donc de créer notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire en suivant le code web suivant.

▷ Voir le code
Code web : [308908](#)

En résumé

Vous savez maintenant lire et écrire un document XML grâce à l'**implémentation Java de l'API DOM**.

Exceptionnellement, il n'y aura pas de TP sur cette technologie. En effet, DOM étant implémenté dans de nombreux langages de programmation, il est impossible pour moi de proposer une correction dans chacun des langages utilisés !

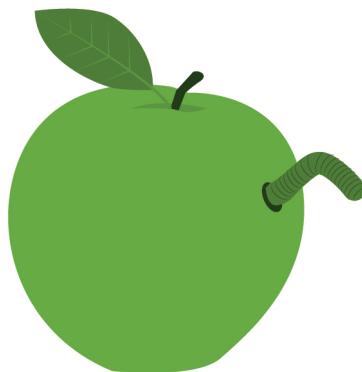
XPath : Introduction à l'API

Difficulté : 

Dans les chapitres précédents, nous avons étudié l'une des technologies permettant d'exploiter les informations présentes dans un document XML : l'**API DOM**. Comme je vous le disais précédemment, dans cette troisième partie du cours, 2 technologies seront abordées. Il nous en reste donc une à voir : **XPath**.

Cette technologie qui repose sur l'écriture d'expressions va nous permettre d'extraire de manière très précise les informations contenues dans un document XML.

Ce premier chapitre sera l'occasion de faire connaissance avec **XPath** en revenant principalement sur le vocabulaire à connaître pour la manipuler.



Qu'est-ce que l'API XPath ?

La petite histoire de XPath

XPath est une technologie qui permet d'extraire des informations (éléments, attributs, commentaires, etc...) d'un document XML via l'écriture d'expressions dont la syntaxe rappelle les expressions rationnelles utilisées dans d'autres langages.

Tout comme DOM, XPath est un standard du W3C et ce depuis sa première version en 1999. Au moment où j'écris ces lignes, la technologie en est à sa deuxième version.

Si XPath n'est pas un langage de programmation en soit, cette technologie fournit tout un vocabulaire pour écrire des expressions permettant d'accéder directement aux informations souhaitées sans avoir à parcourir tout l'arbre XML.

Un peu de vocabulaire

Avant d'étudier de manière plus approfondie comment écrire des expressions XPaths, il convient de revenir sur quelques notions de vocabulaire qui seront indispensables pour bien comprendre la suite du cours.

Pour ce faire, je vous propose de reprendre un document XML que nous avons déjà vu plusieurs fois dans ce cours et l'utiliser pour illustrer les notions que nous allons voir :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</
10      telephone>
11    </telephones>
12  </personne>
13 </repertoire>
```

Reprenez également une illustration de son arbre (voir figure 18.3).

Pour rappel, voici la légende à la figure 18.2.

Parent

Le **parent** d'un noeud est le noeud qui est directement au dessus de lui d'un point de vue hiérarchique. Chaque noeud a au moins un parent.

Par exemple, le noeud **repertoire** est le parent du noeud **personne** qui est lui-même le parent des noeuds **nom**, **prenom** et **telephones**.

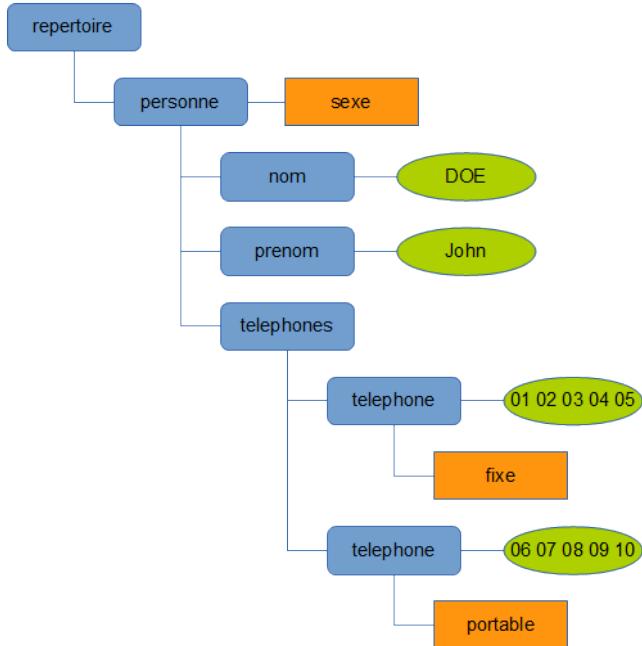


FIGURE 18.1 – Arbre XML d'un document XML

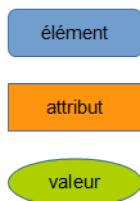


FIGURE 18.2 – Légende de l'arbre XML

Enfant

Un nœud a pour **enfants** tous les noeuds situés un niveau en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité d'enfants.

Par exemple, le nœud **repertoire** a pour enfant le nœud **personne** qui a lui même plusieurs enfants : les noeuds **nom**, **prenom** et **telephones**.

Descendant

Un nœud a pour **descendants** tous les noeuds situés en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité de descendants.

Par exemple, le nœud **repertoire** a pour descendants les nœuds **personne**, **nom**, **prenom** et **telephones**.

Ancêtre

Un nœud a pour **ancêtres** tous les noeuds situés en dessus dans la hiérarchie. Un nœud peut donc avoir plusieurs ancêtres.

Par exemple, le nœud **telephones** a pour ancêtres les nœuds **personne** et **repertoire**.

Frère

Un nœud a pour **frères** tous les noeuds situés au même niveau dans la hiérarchie. Un nœud peut donc avoir une infinité de frères.

Par exemple, le nœud **nom** a pour frères les nœuds **prenom** et **telephones**.

Chemin relatif et chemin absolu

Vous l'aurez compris avec le chapitre précédent, XPath est une technologie qui permet d'extraire des informations d'un document XML via l'écriture d'expressions. Concrètement, ces expressions consistent à décrire le chemin emprunté dans l'arbre XML pour atteindre les données qui nous intéressent.

Reprenons le schéma utilisé jusqu'ici pour illustrer le principe (voir figure 18.3).

Si je veux récupérer par exemple le numéro de fixe voici le chemin à parcourir :

- **Etape 1** : nœud « **repertoire** ».
- **Etape 2** : descendre au nœud enfant « **personne** ».
- **Etape 3** : descendre au nœud enfant « **telephones** ».
- **Etape 4** : descendre au nœud enfant « **telephone** » dont l'attribut est « **fixe** ».

Sans rentrer dans les détails, l'expression XPath correspondante ressemblera à quelque chose comme ça :

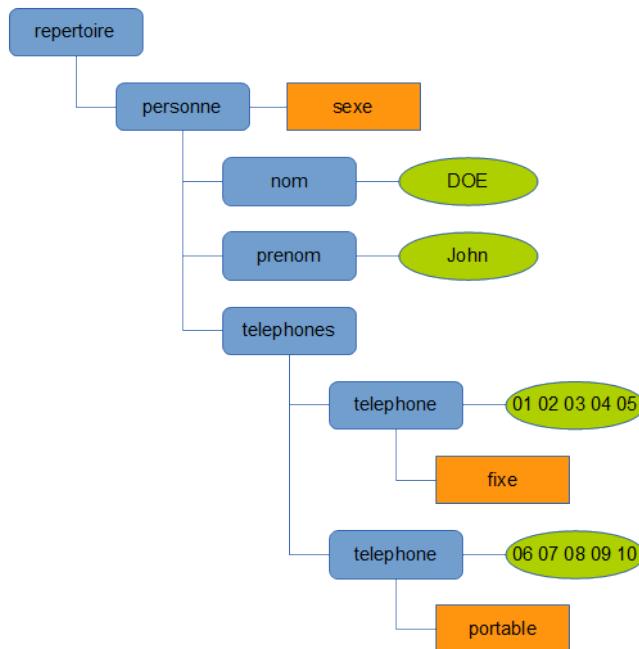


FIGURE 18.3 – Arbre XML d'un document XML

1 | /étape1/étape2/étape3/étape4

Si le principe est toujours le même, il est possible d'exprimer vos chemins de 2 manières :

- Un **chemin relatif**.
- Un **chemin absolu**.

Les chemins absous

Le chemin absolu est le type de chemin que nous avons utilisé dans notre exemple. Le noeud de départ est toujours la racine de l'arbre XML.

Une expression XPath utilisant un chemin absolu est facilement identifiable car elle commence par le caractère « / ».

Bien que nous ayons déjà vu un exemple, je vous propose d'illustrer cette définition par un nouvel exemple dans lequel nous allons récupérer le prénom de la personne décrite dans notre arbre XML :

- **Etape 1** : noeud « **repertoire** ».
- **Etape 2** : descendre au noeud enfant « **personne** ».
- **Etape 3** : descendre au noeud enfant « **prenom** ».

L'expression XPath correspondante ressemblera alors à ça :

1 | /étape1/étape2/étape3

Les chemins relatifs

Si un chemin absolu est un chemin dont le nœud de départ est toujours la racine de l'arbre XML, **un chemin relatif** accepte quant à lui n'importe quel noeud de l'arbre XML comme point de départ.

Une expression XPath utilisant un chemin relatif est facilement identifiable car elle ne commence pas par le caractère « / ».

Comme pour les chemins absolus, je vous propose d'illustrer cette nouvelle définition par un exemple dans lequel nous allons récupérer le prénom de la personne. Dans cet exemple, notre point de départ sera le nœud décrivant le numéro de téléphone portable de John DOE :

- **Etape 1** : nœud « **telephone** » dont l'attribut est « **portable** ».
- **Etape 2** : remonter au nœud parent « **telephones** ».
- **Etape 3** : aller nœud frère « **prenom** ».

L'expression XPath correspondante ressemblera alors à ça :

1 | étape1/étape2/étape3

En résumé

- **XPath** est une technologie qui permet d'extraire des informations d'un document XML via l'écriture de d'expressions.
- **Parent**, **enfant**, **descendant**, **ancêtre** et **frère** sont des notions importantes à comprendre afin de manipuler XPath.
- Les **expressions XPath** peuvent être écrites à l'aide d'un chemin **relatif** ou **absolu**.

XPath : Localiser les données

Difficulté : 

Le chapitre précédent, peu technique, a surtout été l'occasion de faire connaissance avec le vocabulaire utilisé dans XPath. Je préfère vous prévenir, ce nouveau chapitre sera dense !

Nous allons aborder toutes les notions qui vous permettront d'**utiliser XPath** afin d'exploiter facilement vos documents XML. Au programme donc : la dissection détaillée d'une **étape** et un peu de pratique via **EditiX**.



Dissection d'une étape

Dans le chapitre précédent, nous avons vu qu'une expression XPath est en réalité une succession d'étapes. Nous allons maintenant nous intéresser de plus près à ce qu'est une **étape**.

Une étape est décrite par 3 éléments :

- Un **axe**.
- Un **nœud** ou un type de nœud.
- Un ou plusieurs **prédictats** (facultatif).

Avant de voir en détail les valeurs possibles pour ces 3 éléments, je vous propose de revenir très rapidement sur leurs rôles respectifs.

L'axe

L'**axe** va nous permettre de définir le *sens* de la recherche. Par exemple, si l'on souhaite se diriger vers un nœud enfant ou au contraire remonter vers un nœud parent voir un ancêtre.

Le nœud

Ce second élément va nous permettre d'affiner notre recherche en indiquant explicitement le **nom d'un nœud** ou le **type de nœud** dont les informations nous intéressent.

Les prédictats

Comme précisé un peu plus haut, ce dernier élément est *facultatif*. Les **prédictats**, dont le nombre n'est pas limité, agissent comme un filtre et vont nous permettre de gagner en précision lors de nos recherches. Ainsi, grâce aux **prédictats**, il sera par exemple possible de sélectionner les informations à une position précise.

Maintenant que nous savons comment former une étape, il nous reste à apprendre la syntaxe nous permettant de les ordonner et ainsi de pouvoir écrire une étape compatible avec XPath :

1 | axe :: nœud [predicat] [predicat] ... [predicat]

Les axes

Comme nous l'avons vu dans la partie précédente, un **axe** est le premier élément formant une étape. Son rôle est de définir le *sens* de la recherche. Bien évidemment, le choix du sens est structuré par un vocabulaire précis que nous allons étudier maintenant.

Le tableau récapitulatif

Nom de l'axe	Description
ancestor	oriente la recherche vers les ancêtres du noeud courant
ancestor-or-self	oriente la recherche vers le nœud courant et ses ancêtres
attribute	oriente la recherche vers les attributs du noeud courant
child	oriente la recherche vers les enfants du noeud courant
descendant	oriente la recherche vers les descendants du noeud courant
descendant-or-self	oriente la recherche vers le nœud courant et ses descendants
following	oriente la recherche vers les nœuds suivant le noeud courant
following-sibling	oriente la recherche vers les frères suivants du noeud courant
parent	oriente la recherche vers le père du noeud courant
preceding	oriente la recherche vers les nœuds précédent le noeud courant
preceding-sibling	oriente la recherche vers les frères précédents du noeud courant
self	oriente la recherche vers le nœud courant



Pour votre culture générale, sachez qu'il existe également un axe nommé **namespace** qui permet d'orienter la recherche vers un espace de noms. Je l'ai volontairement retiré du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.

Quelques abréviations

Tout au long de notre découverte des axes, des tests de nœuds et des prédictats, nous allons découvrir qu'il est possible d'utiliser des abréviations afin de rendre la syntaxe de nos expressions XPath plus claire et concise.

L'axe child

Pour les axes, il existe une abréviation possible et elle concerne l'axe **child**. En réalité, lorsque l'on souhaite orienter la recherche vers l'axe child, ce n'est pas nécessaire de le préciser. Il s'agit de l'axe par défaut.

Les tests de nœuds

Nous venons donc de voir les différentes valeurs possibles pour sélectionner un axe. Il est donc maintenant temps d'attaquer le second élément composant une étape : le **nom** d'un noeud ou le **type** de noeud.

Nom	Description
nom du nœud	orienté la recherche vers le nœud dont le nom a explicitement été spécifié
*	orienté la recherche vers tous les nœuds
node()	orienté la recherche vers tous les types de nœuds (éléments, commentaires, attributs, etc.)
text()	orienté la recherche vers les nœuds de type texte
comment()	orienté la recherche vers les nœuds de type commentaire

Le tableau récapitulatif



A noter : il existe également d'autres valeurs possibles comme par exemple **processing-instruction()**. Je l'ai volontairement retiré du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.

Quelques exemples

Puisque les prédictats sont facultatifs dans les expressions XPath, je vous propose de voir d'ores et déjà quelques exemples. Pour les exemples, nous allons nous appuyer sur le document XML que nous avons déjà utilisé :

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2  <repertoire>
3      <!-- John DOE -->
4      <personne sexe="masculin">
5          <nom>DOE</nom>
6          <prenom>John</prenom>
7          <adresse>
8              <numero>7</numero>
9              <voie type="impassé">impasse du chemin</voie>
10             <codePostal>75015</codePostal>
11             <ville>PARIS</ville>
12             <pays>FRANCE</pays>
13         </adresse>
14         <telephones>
15             <telephone type="fixe">01 02 03 04 05</telephone>
16             <telephone type="portable">06 07 08 09 10</
17                 telephone>
18         </telephones>
19         <emails>
20             <email type="personnel">john.doe@wanadoo.fr</email>
21             <email type="professionnel">john.doe@societe.com</
22                 email>
23         </emails>
24     </personne>
25 </repertoire>
```

Les chemins absous

Nous allons débuter par des exemples se basant sur l'écriture d'une expression utilisant un chemin absolu.

Dans notre premier exemple, le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- **Etape 1** : descendre au noeud « **repertoire** » .
- **Etape 2** : descendre au noeud « **personne** » .
- **Etape 3** : descendre au noeud « **adresse** » .
- **Etape 4** : descendre au noeud « **pays** » .

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- **Etape 1** : child :: repertoire.
- **Etape 2** : child :: personne.
- **Etape 3** : child :: adresse.
- **Etape 4** : child :: pays.

Ce qui nous donne :

```
1 | /child::repertoire/child::personne/child::adresse/child::pays
```

Il est possible de simplifier l'écriture de cette expression. En effet, comme je l'ai dit dans le chapitre sur les axes, l'axe **child** est celui par défaut, il n'est donc pas nécessaire de le préciser. Ainsi, il est possible de simplifier notre expression de la sorte :

```
1 | /repertoire/personne/adresse/pays
```

Maintenant que vous êtes un peu plus à l'aise avec la syntaxe de XPath, je vous propose de voir un exemple un peu plus exotique. Le but est de trouver maintenant l'expression XPath permettant de **trouver tous les commentaires de notre document XML**.

Dans ce nouvel exemple, une seule étape est en réalité nécessaire et consiste à sélectionner tous les descendants du noeud **racine** qui sont des commentaires.

Tentons maintenant de traduire cette étape sous la forme d'expressions XPath :

- On sélectionne tous les descendants avec l'expression **descendant**.
- On filtre les commentaires avec l'expression **comment()**.

Ce qui nous donne :

```
1 | /descendant::comment()
```

Les chemins relatifs

Après avoir vu quelques exemples d'expressions XPath utilisant des chemins absous, je vous propose de voir un exemple d'une expression utilisant un **chemin relatif**. Dans cet exemple, notre point de départ sera le noeud « **telephones** ». Une fois de plus, le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- **Etape 1 :** remonter au noeud frère « **adresse** ».
- **Etape 2 :** descendre au noeud « **pays** ».

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- **Etape 1 :** preceding-sibling :: adresse.
- **Etape 2 :** pays.

Ce qui nous donne :

```
1 | preceding-sibling::adresse/pays
```

Quelques abréviations

Tout comme pour les axes, il existe quelques abréviations dont je vous conseille d'abuser afin de rendre vos expressions XPath plus lisibles et légères.

L'expression /descendant-or-self :: node() /

Dans nos expressions XPath, il est possible de remplacer l'expression « **/descendant-or-self :: node()** / » par « **//** ».

Ainsi, l'expression :

```
1 | /descendant-or-self::node() /pays
```

peut être simplifiée par :

```
1 | //pays
```

L'expression self :: node()

Notre deuxième abréviation va nous permettre de remplacer l'expression « **/self :: node()** / » par « **.** ».

Ainsi, l'expression :

```
1 | /repertoire/personne/self :: node()
```

peut être simplifiée par :

```
1 | /repertoire/personne/.
```

L'expression parent :: node()

Notre dernière abréviation va nous permettre de remplacer l'expression « **/parent :: node()** / » par « **..** ».

Les prédictats

Nous venons donc de voir deux des trois éléments formant une étape dans une expression XPath. Dans ce chapitre, nous allons donc aborder l'élément manquant : **les prédictats**.

Le tableau récapitulatif

Nom du prédictat	Description
attribute	permet d'affiner la recherche en fonction d'un attribut
count()	permet de compter le nombre de noeuds
last()	permet de sélectionner le dernier noeud d'une liste
position()	permet d'affiner la recherche en fonction de la position d'un noeud



A noter : il existe également d'autres valeurs possibles comme par exemple **name()**, **id()** ou encore **string-length()**. Je les ai volontairement retirées du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.



Un prédictat peut également contenir une expression XPath correspondant à une étape. Nous verrons notamment ce cas dans le TP.

Quelques exemples

Pour les exemples, nous allons continuer de nous appuyer sur le même document XML :

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3     <!-- John DOE -->
4     <personne sexe="masculin">
5         <nom>DOE</nom>
6         <prenom>John</prenom>
7         <adresse>
8             <numero>7</numero>
9             <voie type="impasse">impasse du chemin</voie>
10            <codePostal>75015</codePostal>
11            <ville>PARIS</ville>
12            <pays>FRANCE</pays>
13        </adresse>
14        <telephones>
15            <telephone type="fixe">01 02 03 04 05</telephone>
16            <telephone type="portable">06 07 08 09 10</
                telephone>

```

```
17 |         </telephones>
18 |         <emails>
19 |             <email type="personnel">john.doe@wanadoo.fr</email>
20 |             <email type="professionnel">john.doe@societe.com</
21 |                 email>
22 |         </emails>
23 |     </personne>
23 | </repertoire>
```

Premier exemple

Dans notre premier exemple, le but va être de récupérer le nœud contenant **le numéro de téléphone fixe** de John DOE. Bien évidemment, il existe plusieurs façon d'y arriver. Je vous propose d'utiliser celle qui pousse le moins à réfléchir : nous allons sélectionner tous les descendants du nœud racine et filtrer sur la valeur de l'attribut **type**. Ce qui nous donne :

```
1 | /descendant::*[attribute::type="fixe"]
```

Bien évidemment, cette méthode est à prescrire car elle peut avoir de nombreux effets de bord. Il est possible de procéder autrement en précisant le chemin complet :

```
1 | /repertoire/personne/telephones/telephone[attribute::type="fixe
   "]
```

Terminons ce premier exemple en sélectionnant les numéros de téléphones qui ne sont pas des numéros de téléphones fixes. Une fois de plus, il existe plusieurs façons de procéder. La première, qui *a priori* est la plus simple, consiste à remplacer dans notre expression précédente l'opérateur d'égalité « `=` » par l'opérateur de non égalité « `!=` » :

```
1 | /repertoire/personne/telephones/telephone[attribute::type!="
   fixe"]
```

Une autre méthode consiste à utiliser la fonction **not()** :

```
1 | /repertoire/personne/telephones/telephone[not(attribute::type="
   fixe")]
```

A noter : la double négation nous fait revenir à notre point de départ. En effet, les 2 expressions suivantes sont équivalentes :

```
1 | /repertoire/personne/telephones/telephone[not(attribute::type!=
   "fixe")]
```

```
1 | /repertoire/personne/telephones/telephone[attribute::type="fixe
   "]
```

Deuxième exemple

Après avoir manipulé les attributs, je vous propose maintenant de manipuler les positions. Ainsi, notre deuxième exemple consiste à sélectionner **le premier numéro de téléphone** de John DOE. Commençons par détailler les étapes en français :

- **Etape 1** : descendre au noeud « **repertoire** ».
- **Etape 2** : descendre au noeud « **personne** ».
- **Etape 3** : descendre au noeud « **telephones** ».
- **Etape 4** : sélectionner le premier noeud « **telephone** ».

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- **Etape 1** : repertoire.
- **Etape 2** : personne.
- **Etape 3** : telephones.
- **Etape 4** : telephone[position()=1].

Ce qui nous donne :

```
1 | /repertoire/personne/telephones/telephone [position()=1]
```

Si l'on souhaite maintenant sélectionner le dernier noeud « **téléphone** » de la liste, on modifiera l'expression de la manière suivante :

```
1 | /repertoire/personne/telephones/telephone [last()]
```

Quelques abréviations

Comme pour les axes, nous n'allons voir ici qu'une seule abréviation et elle concerne le prédicat **attribute** qu'il est possible de remplacer par le symbole « **@** ». Ainsi, l'expression :

```
1 | /repertoire/personne/telephones/telephone [attribute::type="fixe"  
      ""]
```

devient :

```
1 | /repertoire/personne/telephones/telephone [@type="fixe"]
```

Un exemple avec EditiX

Pour conclure ce chapitre, je vous propose de voir comment **exécuter une expression XPath avec EditiX**.

Le document XML

Commencez par créer dans EditiX un document XML contenant les informations suivantes :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <repertoire>
4     <!-- John DOE -->
5     <personne sexe="mASCULIN">
6         <nom>DOE</nom>
7         <prenom>John</prenom>
8         <adresse>
9             <numero>7</numero>
10            <voie type="impasse">impasse du chemin</voie>
11            <codePostal>75015</codePostal>
12            <ville>PARIS</ville>
13            <pays>FRANCE</pays>
14        </adresse>
15        <telephones>
16            <telephone type="fixe">01 02 03 04 05</telephone>
17            <telephone type="portable">06 07 08 09 10</
18                telephone>
19        </telephones>
20        <emails>
21            <email type="personnel">john.doe@wanadoo.fr</email>
22            <email type="professionnel">john.doe@societe.com</
23                email>
24        </emails>
25    </personne>
26
27    <!-- Marie POPPINS -->
28    <personne sexe="fEMININ">
29        <nom>POPPINS</nom>
30        <prenom>Marie</prenom>
31        <adresse>
32            <numero>28</numero>
33            <voie type="avenue">avenue de la république</voie>
34            <codePostal>13005</codePostal>
35            <ville>MARSEILLE</ville>
36            <pays>FRANCE</pays>
37        </adresse>
38        <telephones>
39            <telephone type="bureau">04 05 06 07 08</telephone>
40        </telephones>
41        <emails>
42            <email type="professionnel">contact@poppins.fr</
43                email>
44        </emails>
45    </personne>
46 </repertoire>
```

La vue XPath

Afin de pouvoir exécuter des expressions XPath, nous allons devoir afficher la vue dédiée au sein de EditiX. Pour ce faire, vous pouvez sélectionner dans la barre de menu **XML** puis **XPath view** ou encore utiliser le raccourci clavier **Ctrl** + **Shift** + **4**.

La fenêtre visible à la figure 19.1 doit alors apparaître.



FIGURE 19.1 – Vue XPath dans EditiX

Comme vous pouvez le constater, cette vue se compose de plusieurs éléments :

- Un champ dans lequel toutes nos expressions seront écrites.
- 2 boutons permettant de choisir si notre expression utilise un chemin relatif ou absolu.
- Une puce permettant de choisir la version de XPath à utiliser (prenez l'habitude de travailler avec la version 2).
- Des onglets permettant notamment d'afficher les informations sélectionnées par nos expressions.

Exécuter une requête

Dans cet ultime exemple, nous allons sélectionner les nœuds contenant des adresses e-mails professionnelles grâce à l'expression suivante :

```
1 | /repertoire/personne/emails/email[attribute::type="professionnel"]
```

En théorie, nous devrions avoir 2 noeuds sélectionnés. Vérifions tout de suite à la figure 19.2.

Le résultat est bien celui souhaité !

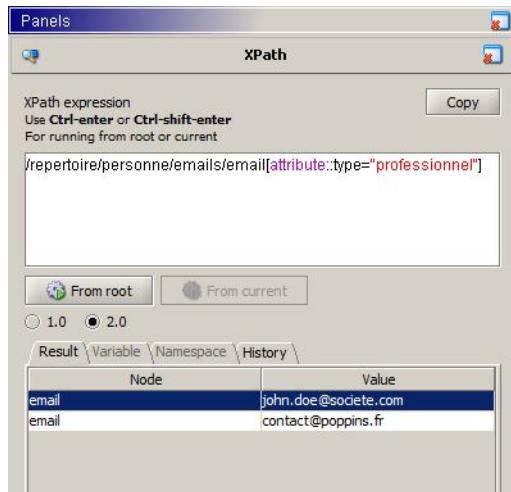


FIGURE 19.2 – Résultat d'une expression XPath dans EditiX

En résumé

- Une **étape** est composée d'un **axe**, d'un **nœud** ou **type de nœud** et d'un ou plusieurs **prédictats**.
- L'axe par défaut est l'axe **child**.
- De nombreuses abréviations permettent de simplifier l'écriture des expressions XPath.

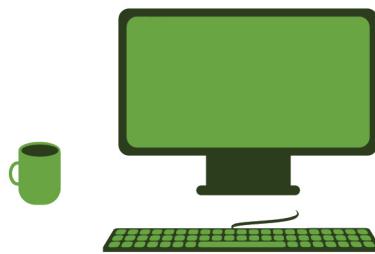
Chapitre 20

TP : des expressions XPath dans un répertoire

Difficulté : 

Après l'apprentissage de l'**API DOM** sans une réelle pratique, il est temps de corriger le tir ! Ce nouveau TP sera donc l'occasion d'écrire plusieurs **expressions XPath** destinées à extraire les informations d'un répertoire contenu dans un document XML.

Bon courage !



L'énoncé

Le document XML

Une fois de plus, c'est avec un répertoire téléphonique que nous allons travailler.

Voici les informations que l'on connaît pour chaque personne :

- Son sexe (homme ou femme).
- Son nom.
- Son prénom.
- Son adresse.
- Un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.).
- Aucune ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici maintenant le document XML qui va nous servir de support :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <adresse>
8       <numero>7</numero>
9       <voie type="impasse">impasse du chemin</voie>
10      <codePostal>75015</codePostal>
11      <ville>PARIS</ville>
12      <pays>FRANCE</pays>
13    </adresse>
14    <telephones>
15      <telephone type="fixe">01 02 03 04 05</telephone>
16      <telephone type="portable">06 07 08 09 10</
17        telephone>
18    </telephones>
19    <emails>
20      <email type="personnel">john.doe@wanadoo.fr</email>
21      <email type="professionnel">john.doe@societe.com</
22        email>
23    </emails>
24  </personne>
25
26  <!-- Marie POPPINS -->
27  <personne sexe="feminin">
28    <nom>POPPINS</nom>
29    <prenom>Marie</prenom>
30    <adresse>
31      <numero>28</numero>
32      <voie type="avenue">avenue de la république</voie>
33      <codePostal>13005</codePostal>
34      <ville>MARSEILLE</ville>
```

```

33         <pays>FRANCE</pays>
34     </adresse>
35     <telephones>
36         <telephone type="professionnel">04 05 06 07 08</
37             telephone>
38     </telephones>
39     <emails>
40         <email type="professionnel">contact@poppins.fr</
41             email>
42     </emails>
43 </personne>
44
45     <!-- Batte MAN -->
46 <personne sexe="masculin">
47     <nom>MAN</nom>
48     <prenom>Batte</prenom>
49     <adresse>
50         <numero>24</numero>
51         <voie type="avenue">impasse des héros</voie>
52         <codePostal>11004</codePostal>
53         <ville>GOTHAM CITY</ville>
54         <pays>USA</pays>
55     </adresse>
56     <telephones>
57         <telephone type="professionnel">01 03 05 07 09</
58             telephone>
59     </telephones>
60 </personne>
61 </repertoire>

```

▷ Copier ce code
Code web : [572860](#)

Les expressions à écrire

Voici donc la liste des expressions XPath à écrire :

- Sélectionner tous les nœuds descendants du deuxième nœud « **personne** ».
- Sélectionner le nœud « **personne** » correspondant au individu ayant au moins 2 numéros de téléphone.
- Sélectionner tous les nœuds « **personne** ».
- Sélectionner le deuxième nœud « **personne** » dont le pays de domiciliation est la **France** .
- Sélectionner tous les nœuds « **personne** » de sexe **masculin** le pays de domiciliation est les **Etats-Unis**.

C'est à vous de jouer !

Une solution

Comme à chaque fois, je vous fais part de ma solution.

Expression n° 1

Le but de cette première expression était de sélectionner tous les noeuds descendants du deuxième noeud « **personne** » :

```
1 | /repertoire/personne[position()=2]/descendant::*
```

Expression n° 2

Le but de cette expression était de sélectionner le noeud « **personne** » correspondant à un individu ayant au moins 2 numéros de téléphone :

```
1 | /repertoire/personne[count(telephones/telephone) > 1]
```

Expression n° 3

Le but de cette troisième expression était de sélectionner tous les noeuds « **personne** » :

```
1 | /repertoire/personne
```

ou encore :

```
1 | //personne
```

Expression n° 4

Le but de cette expression était de sélectionner le deuxième noeud « **personne** » dont le pays de domiciliation est la **France** :

```
1 | /repertoire/personne[adresse/pays="FRANCE"][position()=2]
```

Expression n° 5

Le but de la dernière expression était de sélectionner tous les noeuds « **personne** » de sexe **masculin** le pays de domiciliation est les Etats-Unis :

```
1 | /repertoire/personne[@sexe="masculin"][adresse/pays="USA"]
```

Quatrième partie

Transformez vos documents XML

Chapitre 21

Introduction à XSLT

Difficulté : 

Tout au long de ce tutoriel, nous avons découvert de nombreuses technologies régulièrement utilisées en parallèle d'un document XML. Chaque technologie ayant un but très précis : écrire une définition ou encore exploiter les données contenus dans un document XML, par exemple.

Dans cette quatrième partie, nous allons aborder une nouvelle technologie appelée **XSLT**. Cette technologie va nous permettre de **transformer nos documents XML**, c'est-à-dire, de créer de nouveaux documents à partir des informations contenues dans un document XML.

Comme d'habitude, nous allons débuter en douceur. Ce premier chapitre a donc pour objectif de faire connaissance avec la technologie qui va nous occuper dans les prochains chapitres.



I était un fois...

Qu'est-ce que XSLT ?

La petite histoire du XSLT

XSLT ou **eXtensible Stylesheet Language Transformations** est une technologie qui permet de *transformer les informations* d'un document XML vers un autre type de document comme un autre document XML ou encore une page web. C'est d'ailleurs ce dernier cas que nous aborderons au cours de cette partie.

Comme toutes les technologies que nous avons abordé dans ce tutoriel, **XSLT** est un standard du W3C depuis 1999 pour sa première version et 2007 pour sa seconde version. Comme d'autres technologies que nous avons vu jusqu'ici, les documents XSLT sont écrits à l'aide d'un langage de type XML.

Si XSLT permet de faire les transformations, la technologie s'appuie sur une autre technologie (que nous avons déjà vue ensemble) pour sélectionner des informations à transformer : **XPath**.

Si aviez sauté la partie du tutoriel traitant de XPath ou si vous avez besoin de vous rafraîchir la mémoire, n'hésitez pas à relire la partie 3 du cours !

Comment ça marche ?

Le principe de fonctionnement est assez simple : un **document XSLT** est associé à un **document XML** afin de créer un nouveau document d'une nature différente ou identique (voir figure 21.1).

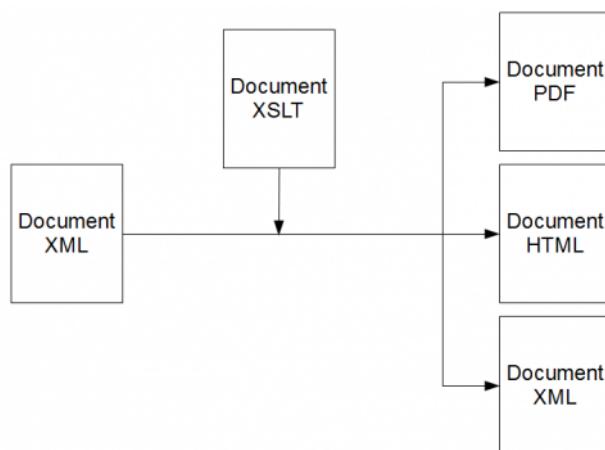


FIGURE 21.1 – Principe d'une transformation XSLT

Structure d'un document XSLT

Maintenant que vous en savez un peu plus sur cette nouvelle technologie, je vous propose de débuter en douceur notre apprentissage du XSLT en découvrant ensemble la structure d'un document.

L'extension du fichier

Comme c'était le cas pour les **DTD** et les **schémas XML**, nous allons prendre l'habitude d'écrire nos **documents XSLT** dans un fichier distinct du document XML dont les données seront transformées.

L'extension portée par les documents XSLT est « **.xsl** ».

Le prologue

Puisque la technologie XSLT utilise un langage de type XML, nous n'allons pas déroger à la règle du **prologue**.

Ainsi, la première ligne d'un document XSLT est :

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

Comme vous pouvez le constater, le prologue est identique à ce que nous avons déjà vu pour les documents XML et les schémas XML. Si vous avez besoin de vous rafraîchir la mémoire sur les différents éléments qui composent ce prologue, je vous invite à relire la partie 1 de ce tutoriel traitant du sujet.

Le corps

Le **corps** d'un fichier XSLT, au même titre qu'un document XML classique est constitué d'un ensemble de **balises** dont l'**élément racine**. Comme c'était déjà le cas pour un schéma XML, l'élément racine d'un document XSLT est imposé.

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

Comme vous pouvez le constater, l'élément racine est donc `<xsl:stylesheet />`.

On remarque la présence de 2 attributs dans cet élément racine. Le premier est le numéro de version que l'on souhaite utiliser. Dans le cadre de ce tutoriel, c'est la version 1 que nous allons utiliser. Bien qu'une version 2 existe, la première version de XSLT reste encore aujourd'hui majoritairement utilisée.

Le second attribut qui est **xmlns :xsl** nous permet de déclarer un **espace de noms**. Une nouvelle fois, si vous n'êtes pas à l'aise avec cette notion, je vous encourage à lire le chapitre dédié aux espaces de nom en annexe de ce tutoriel.

Via la déclaration de cet espace de noms, toutes les balises utilisées dans un document XSLT doivent être préfixées par **xsl** :

La balise output

Directement après l'élément racine, nous allons prendre l'habitude de placer l'élément **<xsl:output />**. Cet élément permet de décrire le document produit à l'issue des différentes transformations. Cet élément prend plusieurs attributs.

L'attribut method

Ce premier attribut permet de préciser le *type* du document produit à l'issue des transformations. 3 valeurs existent :

- **xml** si le document à produire est un document XML.
- **html** si le document à produire est un document HTML.
- **text** si le document à produire est un document texte.



A noter : dans la seconde version de XSLT, il existe une quatrième valeur : **xhtml** si le document à produire est un document xHTML.

L'attribut encoding

Ce second attribut permet de préciser l'*encodage* du document produit à l'issue des transformations. Un grand nombre de valeurs existent pour renseigner la valeur de l'attribut **encoding** :

- UTF-8.
- ISO-8859-1.
- etc.

L'attribut indent

Ce troisième attribut permet d'indiquer si l'on souhaite que le document produit à l'issue des transformations soit *indenté* ou non. 2 valeurs sont possibles :

- Yes.
- No.

Les autres attributs

2 autres attributs peuvent également être utilisés : **doctype-public** et **doctype-system**.

Ces 2 attributs sont utilisés dans le cas où l'on souhaite associer un document DTD au document produit par la transformation XSLT.

L'attribut **doctype-public** permet de renseigner le **FPI** (Formal Public Identifier), c'est-à-dire l'identifiant public de la DTD.

L'attribut **doctype-system**, quant à lui, permet de renseigner l'**URL** de la DTD.

Maintenant que nous avons passé en revue tous les éléments permettant de construire la balise `<xsl:output />`, je vous propose de voir quelques exemples.

Notre premier exemple nous permet de préciser que l'on souhaite produire un document XML :

```
1 | <xsl:output  
2 |     method="xml"  
3 |     encoding="UTF-8"  
4 |     indent="yes" />
```

Voyons maintenant un exemple nous permettant de produire un document HTML 4 :

```
1 | <xsl:output  
2 |     method="html"  
3 |     encoding="UTF-8"  
4 |     doctype-public="-//W3C//DTD HTML 4.01//EN"  
5 |     doctype-system="http://www.w3.org/TR/html4/strict.dtd"  
6 |     indent="yes" />
```

Pour résumer

Pour résumer, voici à quoi ressembleront nos documents XSLT :

```
1 | <!-- le prologue -->  
2 | <?xml version="1.0" encoding="UTF-8" ?>  
3 |  
4 | <!-- l'élément racine -->  
5 | <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999  
6 |   /XSL/Transform">  
7 |  
8 | <!-- l'élément output -->  
9 | <xsl:output  
10 |   method="html"  
11 |   encoding="UTF-8"  
12 |   doctype-public="-//W3C//DTD HTML 4.01//EN"  
13 |   doctype-system="http://www.w3.org/TR/html4/strict.dtd"  
14 |   indent="yes" />  
15 |  
16 | <!-- reste du document XSLT -->  
17 | </xsl:stylesheet>
```

Référencer un document XSLT

Avant de continuer à étudier la création d'un document XSLT, je vous propose de voir comment **référencer une transformation XSLT** dans un document XML.

Où et comment référencer un document XSLT ?

Le référencement d'un document XSLT se fait au niveau du document XML dont les informations seront utilisées au cours de la transformation.

Ce référencement se fait via une petite ligne à placer sous le prologue et avant l'élément racine du document XML :

```
1 | <?xml-stylesheet type="text/xsl" href="mon_document.xsl" ?>
```

Comme vous pouvez le voir, cette balise un peu spéciale possède 2 attributs sur lesquels il est intéressant de revenir rapidement.

L'attribut type

Il permet de définir le *type* du document que nous souhaitons référencer. Dans notre cas, puisqu'il s'agit d'un document XSLT, il convient de renseigner la clef « **text/xsl** ».

L'attribut href

Cet attribut, très connu de ceux qui manipulent régulièrement le HTML et ses variantes, permet d'indiquer l'*URI* du document que l'on souhaite référencer.

Dans mon exemple, il s'agit d'un chemin relatif puisque le document XML et le document XSLT sont situés dans le même dossier.

Pour résumer

Voici à quoi nos fichiers XML ressembleront :

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <?xml-stylesheet type="text/xsl" href="mon_document.xsl" ?>
3 | <racine>
4 |   <!-- contenu du document XML -->
5 | </racine>
```

En résumé

- Un **document XSLT** est associé à un **document XML** afin de créer un nouveau document.

- L'extension portée par les documents XSLT est « **.xsl** ».
- Un **document XSLT** doit être référencé dans le **document XML** qu'il transforme.

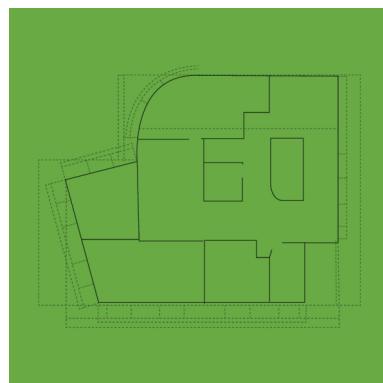
Chapitre 22

Les templates

Difficulté : 

Le chapitre précédent nous a permis de découvrir la technologie **XSLT** et notamment la **structure générale d'un document**. Dans ce nouveau chapitre, nous allons nous attaquer au **corps** d'un document XSLT.

Le **corps d'un document XSLT** est composé d'un ensemble de **templates**. C'est cette notion que nous allons approfondir dans ce chapitre. Nous allons voir, étape par étape, comment écrire un **template** et ainsi sélectionner et exploiter les informations du document XML à transformer.



Introduction aux templates

Maintenant que vous savez qu'un **document XSLT** est composé d'un ensemble de **templates**, je vous propose de rentrer dans le vif du sujet et de découvrir ensemble comment écrire les différents templates qui composeront nos futurs documents XSLT.

Structure d'un template

Un **template** est défini par la balise `<xsl:template />` à laquelle plusieurs attributs peuvent être associés. Dans le cadre de ce tutoriel, nous renseignerons systématiquement l'un des 2 attributs suivants :

- L'attribut **match** permet de renseigner une **expression XPath**. Cette expression XPath permet alors de sélectionner les informations du documents XML auxquelles le template s'applique.
- L'attribut **name** est le nom donné au **template**, permettant de l'identifier de manière unique. Cet attribut est important puisque nous verrons par la suite qu'il n'a pas rare qu'un template en appelle un autre.

```
1 <xsl:template  
2     match="expression XPath"  
3     name="nom du template"  
4 >  
5     <!-- contenu du template -->  
6 </xsl:template>
```



Lorsque nous écrirons nos templates, il est important de renseigner l'un ou l'autre des 2 attributs **match** et **name**, *jamais les deux en même temps*. Nous verrons dans la suite de ce tutoriel que, suivant l'attribut renseigné, le template ne sera pas utilisé de la même façon.

Aller plus loin

Pour aller plus loin, sachez qu'il existe d'autres attributs comme par exemple les attributs **priority** ou **mode** qui permettent respectivement de renseigner une priorité ou un mode de traitement. Dans le cadre de ce tutoriel, nous ne les exploiterons pas et nous nous contenterons d'utiliser les attributs **match** et **name**.

Contenu d'un template

Maintenant que nous avons vu la structure d'un template, il convient d'étudier le cœur d'un template, c'est-à-dire son **contenu**.

Introduction au contenu d'un template

Le **contenu d'un template** permet de définir les transformations à appliquer à l'ensemble des données sélectionnées par l'expression XPath qui lui est attachée. Ce contenu peut-être de différentes natures. Ainsi, il est par exemple possible de simplement remplacer les informations sélectionnées par du texte, ou d'y appliquer des transformations plus complexes à l'aide de nombreuses fonctions que nous étudierons dans ce tutoriel.

Notre premier template

Afin d'illustrer cette définition, je vous propose de voir ensemble un premier exemple dans lequel nous allons créer un document HTML à partir d'un document XML.

Le document XML

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xmlstylesheet type="text/xsl" href="transformation.xsl" ?>
3 <personne>
4     <nom>BATTE</nom>
5     <prenom>Man</prenom>
6 </personne>
```

Comme il est possible de le lire dans ce document XML, à ce dernier est associé le document XSLT **transformation.xsl**. C'est lui qui contient toutes les instructions qui permettent de transformer les données du document XML en un document HTML.

Le document XSLT

Pour notre exemple, voici le contenu de du document **transformation.xsl** :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999
   /XSL/Transform">
3   <xsl:output
4     method="html"
5     encoding="UTF-8"
6     doctype-public="-//W3C//DTD HTML 4.01//EN"
7     doctype-system="http://www.w3.org/TR/html4/strict.dtd"
8     indent="yes" />
9
10  <xsl:template match="/">
11    <html>
12      <head>
13        <title>Mon premier document XSLT</title>
14      </head>
15      <body>
16        <p>Bonjour !</p>
17      </body>
18    </html>
```

```
19 |     </xsl:template>
20 | </xsl:stylesheet>
```

Revenons sur les différents éléments qui composent ce document XSLT. Attardons nous tout d'abord sur la balise `<xsl:output />` dont le contenu ne doit pas vous être inconnu. Les attributs de cette balise nous permettent d'indiquer que nous souhaitons un document HTML 4 dont le contenu sera indenté et encodé en UTF-8.

Regardons maintenant du côté de notre premier template. Nous pouvons nous apercevoir qu'il possède l'attribut « `select` », qui sélectionne toutes les données de notre document XML via l'expression XPath « `/` ».

Revenons maintenant sur la réelle nouveauté de ce document XSLT à savoir, le contenu du template. Ici, il s'agit tout simplement de l'écriture d'un document HTML qui, en réalité, n'exploite absolument pas les données sélectionnées dans le document XML. Ainsi, le document produit suite à l'application de ce template sera toujours le même, peu importe les données fournies par le document XML. Ne vous inquiétez pas, nous verrons comment exploiter les données sélectionnées dans la suite du tutoriel.

Le résultat

Le document HTML produit par la transformation du document XML par le document XSLT est donc le suivant :

```
1  <!DOCTYPE html
2      PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3          html4/strict.dtd">
4
5      <html>
6          <head>
7              <meta http-equiv="Content-Type" content="text/html;
8                  charset=UTF-8">
9              <title>Mon premier document XSLT</title>
10             </head>
11             <body>
12                 <p>Bonjour !</p>
13             </body>
14         </html>
```

Plusieurs choses importantes sont à noter. Tout d'abord, on retrouve bien les informations que nous avions précisées dans la balise `<xsl:output />` de notre document XSLT à savoir les informations sur le Doctype, les informations sur l'encodage ainsi que l'indentation du document.

Les fonctions

Après avoir créé notre premier template dans le chapitre précédent, je vous propose maintenant de nous attaquer aux différentes fonctions que le XSLT met à notre dispo-

sition. Ces fonctions vont nous permettre d'exploiter les données sélectionnées par nos templates afin de procéder à des transformations plus ou moins complexes.

Dans la suite de ce chapitre, le document XML utilisé sera le suivant :

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <?xml-stylesheet type="text/xsl" href="transformation.xsl" ?>
3  <repertoire>
4      <!-- John DOE -->
5      <personne sexe="masculin">
6          <nom>DOE</nom>
7          <prenom>John</prenom>
8          <adresse>
9              <numero>7</numero>
10             <voie type="impassage">impasse du chemin</voie>
11             <codePostal>75015</codePostal>
12             <ville>PARIS</ville>
13             <pays>FRANCE</pays>
14         </adresse>
15         <telephones>
16             <telephone type="fixe">01 02 03 04 05</telephone>
17             <telephone type="portable">06 07 08 09 10</
18                 telephone>
19         </telephones>
20         <emails>
21             <email type="personnel">john.doe@wanadoo.fr</email>
22             <email type="professionnel">john.doe@societe.com</
23                 email>
24         </emails>
25     </personne>
26
27     <!-- Marie POPPINS -->
28     <personne sexe="feminin">
29         <nom>POPPINS</nom>
30         <prenom>Marie</prenom>
31         <adresse>
32             <numero>28</numero>
33             <voie type="avenue">avenue de la république</voie>
34             <codePostal>13005</codePostal>
35             <ville>MARSEILLE</ville>
36             <pays>FRANCE</pays>
37         </adresse>
38         <telephones>
39             <telephone type="professionnel">04 05 06 07 08</
40                 telephone>
41         </telephones>
42         <emails>
43             <email type="professionnel">contact@poppins.fr</
44                 email>
45         </emails>
46     </personne>
47 
```

```
43      <!-- Batte MAN -->
44      <personne sexe="masculin">
45          <nom>MAN</nom>
46          <prenom>Batte</prenom>
47          <adresse>
48              <numero>24</numero>
49              <voie type="avenue">impasse des héros</voie>
50              <codePostal>11004</codePostal>
51              <ville>GOTHAM CITY</ville>
52              <pays>USA</pays>
53          </adresse>
54          <telephones>
55              <telephone type="professionnel">01 03 05 07 09</
56                  telephone>
57          </telephones>
58      </personne>
59  </repertoire>
```

▷ Copier ce code
Code web : 996683

Afin de gagner en clarté, je ne vais pas réécrire à chaque fois la totalité du document XSLT utilisé. Je me contenterai d'écrire uniquement l'unique template que notre document contiendra. Ainsi, la structure générale du document **transformation.xsl** est la suivante :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999
   /XSL/Transform">
3      <xsl:output
4          method="html"
5          encoding="UTF-8"
6          doctype-public="-//W3C//DTD HTML 4.01//EN"
7          doctype-system="http://www.w3.org/TR/html4/strict.dtd"
8          indent="yes" />
9
10     <!-- template -->
11
12  </xsl:stylesheet>
```

La fonction value-of

Syntaxe et explications

La fonction **<xsl:value-of />**, que l'on peut traduire en français par « **valeur de** », nous permet d'extraire la *valeur d'un élément XML ou la valeur de ses attributs*.

Cette fonction possède un attribut **select** auquel il convient de renseigner une expression XPath permettant alors de sélectionner les informations à extraire :

```
1 | <xsl:value-of select="expression XPath" />
```

Exemple

À titre d'exemple, je vous propose d'écrire un template qui va afficher le type du numéro de téléphone et le numéro en lui même de Marie POPPINS :

```
1 | <xsl:template match="/">
2 |   <html>
3 |     <head>
4 |       <title>Test de la fonction value-of</title>
5 |     </head>
6 |     <body>
7 |       <p>Type du numéro : <xsl:value-of select="
8 |           repertoire/personne[nom='POPPINS']/telephones/
9 |           telephone/@type" /></p>
10 |      <p>Numéro : <xsl:value-of select="repertoire/
11 |          personne[nom='POPPINS']/telephones/telephone" />
12 |      </p>
13 |   </body>
14 | </html>
15 | </xsl:template>
```

Comme vous pouvez le constater, notre template contient 2 fonctions `<xsl:value-of />`. La première extrait l'attribut « **type** » de l'unique numéro de téléphone de Marie POPPINS, tandis que le seconde extrait la valeur de l'élément à proprement parler, le tout grâce à des expressions XPath.

Le document HTML produit est alors le suivant :

```
1 | <!DOCTYPE html
2 | PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3 |   html4/strict.dtd">
4 | <html>
5 |   <head>
6 |     <meta http-equiv="Content-Type" content="text/html;
7 |       charset=UTF-8">
8 |
9 |   <title>Test de la fonction value-of</title>
10 |  </head>
11 |  <body>
12 |    <p>Type du numéro : professionnel</p>
13 |    <p>Numéro : 04 05 06 07 08</p>
14 |  </body>
15 | </html>
```

La fonction for-each

Syntaxe et explications

La fonction `<xsl:for-each />`, que l'on peut traduire en français par « **pour chaque** » est une fonction qui permet de boucler sur un ensemble d'éléments.

Par exemple, si l'on souhaite appliquer une transformation à l'ensemble des numéros de téléphone d'une personne, nous allons tous les sélectionner à l'aide d'une expression XPath, puis, grâce à la fonction `<xsl:for-each />`, il sera possible de parcourir l'ensemble des résultats obtenus pour y appliquer une transformation particulière.

Comme pour la fonction `<xsl:value-of />`, la fonction `<xsl:for-each />` possède un attribut **select** auquel il convient de renseigner une expression XPath permettant alors de sélectionner les informations à extraire :

```
1 | <xsl:for-each select="expression XPath" />
```

Exemple

Pour illustrer l'utilisation de la fonction `<xsl:for-each />`, je vous propose d'écrire un template qui va afficher pour chaque adresse e-mail de John DOE son type et son contenu :

```
1 | <xsl:template match="/">
2 |   <html>
3 |     <head>
4 |       <title>Test de la fonction for-each</title>
5 |     </head>
6 |     <body>
7 |       <xsl:for-each select="repertoire/personne[nom='DOE'
8 |           ']//emails[email]">
9 |         <p>Type de l'adresse e-mail : <xsl:value-of
10 |             select="@type" /></p>
11 |         <p>adresse e-mail : <xsl:value-of select="."
12 |             /></p>
13 |       </xsl:for-each>
14 |     </body>
15 |   </html>
16 | </xsl:template>
```

Comme vous pouvez le constater, notre template contient une fonction `<xsl:for-each />` dont le rôle est de sélectionner l'ensemble des adresses e-mails de John DOE. A l'intérieur de cette fonction, nous sommes alors capable d'exploiter individuellement les différentes adresses grâce à la fonction `<xsl:value-of />` que nous avons vu précédemment.

Le document HTML produit est alors le suivant :

```
1 | <!DOCTYPE html
```

```
2 PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3   html4/strict.dtd">
4 <html>
5   <head>
6     <meta http-equiv="Content-Type" content="text/html;
7       charset=UTF-8">
8     <title>Test de la fonction for-each</title>
9   </head>
10  <body>
11    <p>Type de l'adresse e-mail : personnel</p>
12    <p>adresse e-mail : john.doe@wanadoo.fr</p>
13    <p>Type de l'adresse e-mail : professionnel</p>
14    <p>adresse e-mail : john.doe@societe.com</p>
15  </body>
16 </html>
```

La fonction sort

Syntaxe et explications

La fonction `<xsl:sort />` est une fonction qui permet de trier un ensemble d'éléments par ordre croissant ou décroissant.

Puisque l'on souhaite trier un ensemble d'éléments, la fonction `<xsl:sort />` est généralement utilisée au sein de la fonction `<xsl:for-each />` que nous venons de voir. Elle possède au moins un attribut **select** auquel il convient de renseigner une expression XPath permettant alors de sélectionner les informations à trier.

La fonction `<xsl:sort />` accepte également d'autres attributs qui sont cependant optionnels :

- **Order** qui accepte les valeurs **ascending** (croissant) et **descending** (décroissant) permet de préciser le sens du tri. Par défaut, c'est un tri croissant qui est effectué.</puce>
- **Case-order** qui accepte les valeurs **upper-first** (les majuscules d'abord) et **lower-first** (les minuscules d'abord) permet de préciser qui des majuscules et des minuscules est prioritaire pour effectuer les tri.</puce>
- **Data-type** qui accepte les valeurs **text** (texte) et **number** (nombre) permet préciser si les données à trier sont des nombres ou du texte.</puce>
- **Lang** qui accepte pour valeur le code d'une langue (fr pour la langue française, es pour la langue espagnole, it pour l'italien, etc.) qui permet de préciser la langue des éléments à trier. Cet attribut est notamment utile afin de trier des éléments dont la langue possède quelques caractères spécifiques comme par exemple le norvégien, le suédois, etc.</puce>

```
1 <xsl:sort select="expression XPath" order="ascending|descending
2   " case-order="upper-first|lower-first" data-type="text|
3     number" lang="fr|es|it|..." />
```

Exemple

Pour illustrer l'utilisation de la fonction `<xsl:sort />`, je vous propose d'écrire un template qui va afficher le nom et le prénom des personnes contenues dans notre document XML, par ordre alphabétique :

```
1 <xsl:template match="/">
2     <html>
3         <head>
4             <title>Test de la fonction sort</title>
5         </head>
6         <body>
7             <xsl:for-each select="repertoire/personne">
8                 <xsl:sort select="nom"/>
9                 <xsl:sort select="prenom"/>
10                <p><xsl:value-of select="nom" />#160;<xsl:
11                    value-of select="prenom" /></p>
12            </xsl:for-each>
13        </body>
14    </html>
15 </xsl:template>
```

Notre template contient une fonction `<xsl:for-each/>` permettant de boucler sur toutes les personnes contenu dans notre document XML. Il contient également deux fonctions `<xsl:sort />` permettant de trier par ordre alphabétique de nom et prénom l'ensemble des personnes. À noter : le caractère « ` ` » est un caractère qui une fois la transformation effectuée sera simplement remplacé par un espace dans notre document HTML.

Le document HTML produit est le suivant :

```
1 <!DOCTYPE html
2     PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3         html4/strict.dtd">
4 
5 <html>
6     <head>
7         <meta http-equiv="Content-Type" content="text/html;
8             charset=UTF-8">
9         <title>Test de la fonction sort</title>
10    </head>
11    <body>
12        <p>DOE#160;John</p>
13        <p>MAN#160;Batte</p>
14        <p>POPPINS#160;Marie</p>
15    </body>
16 </html>
```

La fonction if

Syntaxe et explications

La fonction `<xsl:if />` est une fonction qui permet de conditionner une transformation. Par exemple, grâce à cette fonction, il sera possible de n'appliquer une transformation qu'aux personnes de sexe masculin.

Cette fonction possède un attribut **test** auquel il convient de renseigner la condition. Cette condition peut-être la comparaison d'une chaîne de caractères ou de nombres.

```
1 | <xsl:if test="test de comparaison" />
```

Les tests

En XSLT, il existe plusieurs types de tests disponibles. Le tableau suivant les résume et les explique :

Condition	Explication
<code>a = b</code>	Cette condition vérifie que la valeur de l'élément a est égale à la valeur de l'élément b .
<code>not(a = b)</code>	Cette condition vérifie que la valeur de l'élément a n'est pas égale à la valeur de l'élément b .
<code>a = b</code>	Cette condition vérifie que la valeur de l'élément a est égale à la valeur de l'élément b .
<code>a &lt; b</code>	Le symbole <code>&lt;</code> (lower than) traduit en réalité le symbole <code><</code> . La condition est donc en réalité la suivante : a <code><</code> b . Cette condition vérifie que la valeur de l'élément a est strictement inférieure à la valeur de l'élément b .
<code>a &lt;= b</code>	Le symbole <code>&lt;</code> (lower than) traduit en réalité le symbole <code><</code> . La condition est donc en réalité la suivante : a <code><=</code> b . Cette condition vérifie que la valeur de l'élément a est inférieure ou égale à la valeur de l'élément b .
<code>a &gt; b</code>	Le symbole <code>&gt;</code> (greater than) traduit en réalité le symbole <code>></code> . La condition est donc en réalité la suivante : a <code>></code> b . Cette condition vérifie que la valeur de l'élément a est strictement supérieure à la valeur de l'élément b .
<code>a &gt;= b</code>	Le symbole <code>&gt;</code> (greater than) traduit en réalité le symbole <code>></code> . La condition est donc en réalité la suivante : a <code>></code> b . Cette condition vérifie que la valeur de l'élément a est supérieure ou égale à la valeur de l'élément b .

Il est également possible d'effectuer plusieurs tests à la fois. Ainsi, nous pouvons :

- Vérifier qu'un premier test est vrai **ET** qu'un second l'est également.</puce>
- Vérifier qu'un premier test est vrai **OU** qu'un second l'est.</puce>

Dans un test, pour traduire la notion de **ET**, on utilise le mot anglais **and**, tandis que pour traduire la notion de **OU**, on utilise sa traduction anglaise **or**.

Voyons quelques exemples afin d'illustrer toutes ces notions. Par exemple, si l'on souhaite vérifier que la valeur d'un élément a est strictement supérieure à un élément b **ET** un élément c, on écrira :

```
1 | <xsl:if test="a > b and a > c">
2 |     <!-- contenu de la condition -->
3 | </xsl:if>
```

Voyons la condition à écrire si l'on souhaite vérifier que la valeur d'un élément a est égale à la valeur d'un élément b **OU** égale à la valeur d'un élément c :

```
1 | <xsl:if test="a = b or a = c">
2 |     <!-- contenu de la condition -->
3 | </xsl:if>
```

Exemple

Pour illustrer l'utilisation de la fonction `<xsl:if />`, je vous propose d'écrire un template qui va afficher le nom et le prénom des personnes de sexe masculin contenues dans notre document XML :

```
1 | <xsl:template match="/">
2 |     <html>
3 |         <head>
4 |             <title>Test de la fonction if</title>
5 |         </head>
6 |         <body>
7 |             <xsl:for-each select="repertoire/personne">
8 |                 <xsl:if test="@sexe = 'masculin'">
9 |                     <p><xsl:value-of select="nom" />&#160;<xsl:
10 |                         value-of select="prenom" /></p>
11 |                 </xsl:if>
12 |             </xsl:for-each>
13 |         </body>
14 |     </html>
15 | </xsl:template>
```

Notre template parcourt donc l'ensemble des personnes contenues dans notre document XML grâce à la fonction `<xsl:for-each/>`. Grâce à la fonction `<xsl:if/>`, il est possible d'appliquer la suite de la transformation uniquement aux personnes de sexe masculin.

Le document HTML produit est le suivant :

```
1 | <!DOCTYPE html
2 | PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3 |     html4/strict.dtd">
4 |
5 | <html>
```

```
4   <head>
5     <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8">
6     <title>Test de la fonction if</title>
7   </head>
8   <body>
9     <p>DOE&nbsp;John</p>
10    <p>MAN&nbsp;Batte</p>
11  </body>
12 </html>
```

La fonction choose

Syntaxe et explications

La fonction `<xsl:choose />`, tout comme la fonction `<xsl:id />`, permet de conditionner des transformations. En réalité, la fonction `<xsl:choose />` ne s'utilise pas toute seule. En effet, elle permet plusieurs conditions. Ainsi, dans le cas où la première condition n'est pas remplie, la seconde va être testée, puis la troisième, etc. Dans le cas où aucune condition n'est remplie, un cas par défaut peut-être prévu.

Les conditions incluses dans la fonction `<xsl:choose />` s'expriment grâce à la fonction `<xsl:when />`. Le cas par défaut s'exprime quant à lui grâce à la fonction `<xsl:otherwise />`.

La fonction `<xsl:when />` possède un attribut **test** auquel il convient de renseigner une condition.

```
1 <xsl:choose>
2   <xsl:when test="test de comparaison">
3     <!-- suite de la transformation -->
4   </xsl:when>
5   <xsl:when test="test de comparaison">
6     <!-- suite de la transformation -->
7   </xsl:when>
8   <xsl:otherwise>
9     <!-- suite de la transformation -->
10  </xsl:otherwise>
11 </xsl:choose>
```

Le nombre de fonction `<xsl:when />` varie en fonction du nombre de conditions que vous souhaitez tester.

Exemple

Pour illustrer l'utilisation de la fonction `<xsl:choose />`, je vous propose d'écrire un template qui suivant le nom des personnes contenues dans le document XML, affichera

une phrase personnalisée. Ainsi, pour John DOE, la phrase à afficher sera « **Bonjour John !** », pour Marie POPPINS, la phrase à afficher sera « **Quel beau sac !** ». Dans tous les autres cas, la phrase à afficher sera « **Qui êtes-vous ?** ».

```
1 <xsl:template match="/">
2     <html>
3         <head>
4             <title>Test de la fonction sort</title>
5         </head>
6         <body>
7             <xsl:for-each select="repertoire/personne">
8                 <xsl:choose>
9                     <xsl:when test="nom = 'DOE'">
10                         <p>Bonjour John !</p>
11                     <xsl:when test="nom = 'POPPINS'">
12                         <p>Quel beau sac !</p>
13                     <xsl:otherwise>
14                         <p>Qui êtes-vous ?</p>
15                     </xsl:otherwise>
16                 </xsl:choose>
17             </xsl:for-each>
18         </body>
19     </html>
20 </xsl:template>
```

Puisque les personnes de notre document XML seront traitées dans leur ordre d'apparition de notre document XML, les phrases seront affichées dans cet ordre dans le document HTML produit :

- Bonjour John!</puce>
- Quel beau sac!</puce>
- Qui êtes-vous?</puce>

C'est effectivement le cas lorsqu'on regarde le document HTML produit :

```
1 <!DOCTYPE html
2     PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3         html4/strict.dtd">
4
5 <html>
6     <head>
7         <meta http-equiv="Content-Type" content="text/html;
8             charset=UTF-8">
9         <title>Test de la fonction sort</title>
10    </head>
11    <body>
12        <p>Bonjour John !</p>
13        <p>Quel beau sac !</p>
14        <p>Qui &ecirc;tes-vous ?</p>
15    </body>
16 </html>
```

La fonction apply-templates

Syntaxe et explications

La fonction `<xsl:apply-templates />`, permet de continuer la transformation des éléments enfants d'un template.

Dans le cadre de ce tutoriel, nous allons uniquement considérer l'attribut **select** de la fonction `<xsl:apply-templates />` qu'il convient de renseigner à l'aide d'une expression XPath.

```
1 | <xsl:apply-templates select="expression XPath" />
```

Exemple

Pour illustrer l'utilisation de la fonction `<xsl:apply-templates />`, je vous propose de décortiquer ensemble le template.

Tout d'abord, considérons un document XML un peu plus simple que celui que nous utilisions jusqu'à maintenant :

```
1 | <?xml version="1.0" encoding="UTF-8" ?>
2 | <?xmlstylesheet type="text/xsl" href="transformation.xsl" ?>
3 | <repertoire>
4 |     <!-- John DOE -->
5 |     <personne sexe="masculin">
6 |         <nom>DOE</nom>
7 |         <prenom>John</prenom>
8 |     </personne>
9 |
10 |    <!-- Marie POPPINS -->
11 |    <personne sexe="feminin">
12 |        <nom>POPPINS</nom>
13 |        <prenom>Marie</prenom>
14 |    </personne>
15 |
16 |    <!-- Batte MAN -->
17 |    <personne sexe="masculin">
18 |        <nom>MAN</nom>
19 |        <prenom>Batte</prenom>
20 |    </personne>
21 | </repertoire>
```

Considérons alors les templates suivants :

```
1 | <xsl:template match="/">
2 |     <html>
3 |         <head>
4 |             <title>Test de la fonction apply-templates</title>
5 |         </head>
```

```
6      <body>
7          <xsl:apply-templates select="repertoire/personne[
8              nom='POPPINS']" />
9      </body>
10     </html>
11 </xsl:template>
12 <xsl:template match="nom">
13     <p><xsl:value-of select=". . ."/></p>
14 </xsl:template>
15 <xsl:template match="prenom">
16     <p><xsl:value-of select=". . ."/></p>
17 </xsl:template>
```

A l'exécution de la transformation XSLT, le premier template à être appelé est le template dont l'expression XPath capture la racine de notre document XML. Dans ce template, la ligne `<xsl:apply-templates select="repertoire/personne[nom='POPPINS']" />` permet d'indiquer que l'on souhaite continuer la transformation uniquement avec l'élément `<personne />` correspondant à **Marie POPPINS** ainsi que ses fils à savoir les éléments `<nom />` et `<prenom />`. L'élément `<nom />` va donc être transformé grâce au second template écrit dans notre document XSLT puisque son expression XPath le capture. Finalement, en suivant la même logique, l'élément `<prenom />` va, quant à lui, être transformé grâce au dernier template de notre document XSLT.

Le document HTML produit est alors le suivant :

```
1 <!DOCTYPE html
2   PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3     html4/strict.dtd">
4 <html>
5   <head>
6     <meta http-equiv="Content-Type" content="text/html;
7       charset=UTF-8">
8     <title>Test de la fonction sort</title>
9   </head>
10  <body>
11    <p>POPPINS</p>
12    <p>Marie</p>
13  </body>
14 </html>
```

Afin de bien comprendre le mécanisme, je vous propose maintenant de jeter un coup d'œil aux templates suivant :

```
1 <xsl:template match="/">
2   <html>
3     <head>
4       <title>Test de la fonction apply-templates</title>
5     </head>
6     <body>
```

```

7      <xsl:apply-templates select="repertoire/personne[  

8          nom='POPPINS']/nom" />  

9      </body>  

10     </html>  

11 </xsl:template>  

12 <xsl:template match="nom">  

13     <p><xsl:value-of select=". "/></p>  

14 </xsl:template>  

15 <xsl:template match="prenom">  

16     <p><xsl:value-of select=". "/></p>  

17 </xsl:template>  

18 
```

Comme pour notre premier exemple, à l'exécution de la transformation XSLT, le premier template à être appelé est le template dont l'expression XPath capture la racine de notre document XML. Dans ce template, la ligne `<xsl:apply-templates select="repertoire/personne[nom='POPPINS']/nom" />` permet d'indiquer que l'on souhaite continuer la transformation uniquement avec l'élément `<nom />` de la `<personne />` correspondant à **Marie POPPINS**. L'élément `<nom />` va donc être transformé grâce au second template de notre document XSLT. Le dernier template ne sera quant à lui jamais appelé.

Le document HTML produit est alors le suivant :

```

1 <!DOCTYPE html  

2     PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/  

3         html4/strict.dtd">  

4 <html>  

5     <head>  

6         <meta http-equiv="Content-Type" content="text/html;  

7             charset=UTF-8">  

8  

9         <title>Test de la fonction sort</title>  

10    </head>  

11    <body>  

12        <p>POPPINS</p>  

13    </body>  

14 </html>
```

En résumé

- Les **templates** possèdent systématiquement l'un des 2 attributs : **match** ou **name**.
- Les **templates** se construisent à l'aide de nombreuses fonctions.

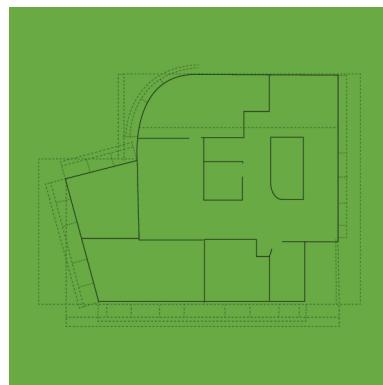
Chapitre 23

Les templates : aller plus loin

Difficulté : 

Dans le chapitre précédent, nous avons vu un grand nombre de fonctions à utiliser dans nos **templates**. Ne nous arrêtons pas en si bon chemin, il nous reste encore quelques fonctions à voir ! Ces nouvelles fonctions seront également l'occasion d'aborder de nouveaux concepts à savoir, les **variables** et les **paramètres**.

Ce nouveau chapitre sera également l'occasion d'apprendre à créer et exécuter une **transformation XSLT dans EditiX**.



Les variables et la fonction call-template

Au cours de ce chapitre, je vous propose d'aborder une nouvelle notion bien connue en informatique et plus précisément dans les langages de programmations : les **variables**. Ce chapitre sera également l'occasion d'une mise en pratique de cette nouvelle notion à travers l'appréhension d'une nouvelle fonction pour nos template : **call-template**.

Les variables

Définition

En XSLT, comme dans de nombreuses technologies informatiques et plus précisément dans les langages de programmation, il existe une notion importante que l'on appelle les **variables**.

Une **variable** est une information que l'on stocke dans la mémoire de l'ordinateur afin de pouvoir la réutiliser. Elle possède un identifiant unique qui est son **nom**. Ainsi, plusieurs variables ne peuvent pas avoir le même nom. L'information stockée par une variable est sa **valeur**.

Dans nos documents XSLT, les variables doivent être déclarées au sein d'un template et ont une portée dite *locale*, c'est-à-dire qu'une variable ne peut être utilisée que dans le template dans lequel elle est déclarée. Par exemple, si une variable portant le nom « **maVariable** » est déclarée dans un premier template, les autres templates du document XSLT ne la connaissent pas.

Déclaration et utilisation

Afin de déclarer une variable, il convient d'utiliser la fonction `<xsl:variable />`. Cette fonction accepte deux attributs :

- **Name** : c'est le nom que l'on souhaite donner à notre variable, ce nom va nous permettre de l'identifier de manière unique.
- **Select** : cet attribut optionnel accepte soit une expression XPath qui permet de sélectionner la valeur que l'on souhaite stocker dans notre variable, soit une chaîne de caractères.

La syntaxe pour déclarer une variable est donc la suivante :

```
1 | <xsl:variable name="nom_de_la_variable" select="expression  
      XPath|chaîne de caractères" />
```

Pour pouvoir utiliser une variable et récupérer la valeur qu'elle contient, il convient de faire précéder son nom par le symbole « **\$** ».

Exemple

Afin d'illustrer ce que nous venons d'apprendre sur les variables, je vous propose de voir ensemble quelques exemples. Dans notre premier exemple, nous allons créer une variable qui contiendra une chaîne de caractères et nous l'afficherons dans une balise paragraphe d'une document HTML :

```

1 | <xsl:template match="/">
2 |   <xsl:variable name="couleur" select="'rouge'" />
3 |
4 |   <html>
5 |     <head>
6 |       <title>Test des variables</title>
7 |     </head>
8 |     <body>
9 |       <p><xsl:value-of select="$couleur" /></p>
10 |     </body>
11 |   </html>
12 | </xsl:template>
```

Comme vous pouvez le constater, dans ce template, nous déclarons une variable qui s'appelle « **couleur** » et qui contient la chaîne de caractères « **rouge** » :

```
1 | <xsl:variable name="couleur" select="'rouge'" />
```

Puisque la valeur « **rouge** » est une chaîne de caractères, il est impératif de la mettre entre ' (guillemets simples) afin de ne pas la confondre avec une expression XPath.

Pour pouvoir afficher la valeur contenue dans la variable dans notre document HTML, il suffit simplement de faire précéder son nom par le symbole \$:

```
1 | <xsl:value-of select="$couleur" />
```

Le document HTML produit par notre transformation est le suivant :

```

1 | <!DOCTYPE html
2 |   PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3 |     html4/strict.dtd">
4 |
5 |   <html>
6 |     <head>
7 |       <meta http-equiv="Content-Type" content="text/html;
8 |         charset=UTF-8">
9 |       <title>Test de la fonction sort</title>
10 |     </head>
11 |     <body>
12 |       <p>rouge</p>
13 |     </body>
14 |   </html>
```

A noter : il existe une autre façon de définir la valeur d'une variable lorsque celle-ci ne provient pas d'une expression XPath :

```
1 | <xsl:variable name="nom">chaîne de caractères</xsl:variable>
```

Ainsi, pour le même document HTML final, nous aurions pu écrire le template suivant :

```
1 <xsl:template match="/">
2     <xsl:variable name="couleur">rouge</xsl:variable>
3
4     <html>
5         <head>
6             <title>Test des variables</title>
7         </head>
8         <body>
9             <p><xsl:value-of select="$couleur" /></p>
10        </body>
11    </html>
12 </xsl:template>
```

Pour notre second exemple, nous allons construire un template dans la valeur des variables sera alimentées par une expression XPath. Pour illustrer notre exemple, nous allons exploiter le document XML suivant :

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <?xmlstylesheet type="text/xsl" href="transformation.xsl" ?>
3 <repertoire>
4     <!-- John DOE -->
5     <personne sexe="masculin">
6         <nom>DOE</nom>
7         <prenom>John</prenom>
8     </personne>
9
10    <!-- Marie POPPINS -->
11    <personne sexe="feminin">
12        <nom>POPPINS</nom>
13        <prenom>Marie</prenom>
14    </personne>
15
16    <!-- Batte MAN -->
17    <personne sexe="masculin">
18        <nom>MAN</nom>
19        <prenom>Batte</prenom>
20    </personne>
21 </repertoire>
```

Dans cette exemple, nous allons déclarer 3 variables :

- La variable **sexe** qui contiendra le sexe d'une personne.
- La variable **nom** qui contiendra le nom d'une personne.
- La variable **prenom** qui contiendra le prénom d'une personne.

Le but de notre template est alors d'afficher le sexe, le prénom et le nom de chaque personne contenue dans le document XML :

```
1 <xsl:template match="/">
2     <html>
```

```

3      <head>
4          <title>Test des variables</title>
5      </head>
6      <body>
7          <xsl:for-each select="repertoire/personne">
8              <xsl:variable name="sexe" select="@sexe" />
9              <xsl:variable name="nom" select="nom" />
10             <xsl:variable name="prenom" select="prenom" />
11
12             <p><xsl:value-of select="$nom"/> &#160; <xsl:
13                 value-of select="$prenom"/> : <xsl:value-of
14                 select="$sexe"/></p>
15         </xsl:for-each>
16     </body>
17     </html>
18 </xsl:template>

```

Dans ce template, nous parcourons l'ensemble des personnes contenues dans le document XML à l'aide de la fonction `<xsl:for-each />`, puis nous déclarons 3 variables dont les valeurs sont renseignées à l'aide d'expressions XPath. Grâce à la fonction `<xsl:value-of />`, il est ensuite très simple d'afficher le contenu de la variable.

La fonction call-template

Maintenant que vous maîtrisez les variables, il est grand temps de mettre en pratique vos connaissances ! En effet, dans les exemples que nous avons vu jusqu'à maintenant, l'utilisation des variables n'était pas pertinente, et nous pouvons facilement arriver au même résultat sans les utiliser. La fonction que nous allons voir maintenant va nous permettre d'utiliser les variables dans des cas utiles.

Définition : première partie

La fonction `<xsl:call-template />` permet d'appeler un autre template au sein d'un template. Il est possible de l'appeler grâce à son nom. Cette fonction accepte donc un attribut « **name** » qui permet d'identifier le template à appeler.

Jusqu'à maintenant, nous avions renseigné l'attribut « **select** » de nos templates grâce à une expression XPath. Les templates appelés grâce à la fonction `<xsl:call-template />` doivent être identifiés par un nom. Il ne convient donc plus de renseigner l'attribut « **select** », mais l'attribut « **name** » qui permet d'identifier de manière unique le template.

Voici alors la syntaxe d'un template à appeler :

```

1 <xsl:template name="nom_du_template">
2     <!-- contenu -->
3 </xsl:template>

```

Pour l'appeler, il convient donc d'utiliser la fonction `<xsl:call-template />` et de renseigner le nom du template à appeler :

```
1 | <xsl:template select="/">
2 |     <xsl:call-template name="nom_du_template" />
3 | </xsl:template>
```

Afin d'illustrer ce premier aperçu de la fonction `<xsl:call-template />`, je vous propose un premier exemple. Dans celui-ci, nous allons écrire un premier template qui capture la racine d'un document XML et qui écrit la structure d'un document HTML. Ce même template appellera un second template grâce à la fonction que nous venons de voir afin que celui-ci peuple le contenu de l'élément `<body />` de notre document HTML à produire.

Voici les templates à écrire :

```
1 | <xsl:template match="/">
2 |     <html>
3 |         <head>
4 |             <title>Test de la fonction call-template</title>
5 |         </head>
6 |         <body>
7 |             <xsl:call-template name="body" />
8 |         </body>
9 |     </html>
10 | </xsl:template>
11 |
12 | <xsl:template name="body">
13 |     <p>Contenu de la page HTML</p>
14 | </xsl:template>
```

Comme vous pouvez le constater, notre premier template appelle un second template identifié par le nom « **body** » grâce à la fonction `<xsl:call-template />` que nous venons de voir.

Le document HTML produit est bien conforme à nos attentes :

```
1 | <!DOCTYPE html
2 | PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3 |     html4/strict.dtd">
4 | <html>
5 |     <head>
6 |         <meta http-equiv="Content-Type" content="text/html;
7 |             charset=UTF-8">
8 |         <title>Test des variables</title>
9 |     </head>
10 |    <body>
11 |        <p>Contenu de la page HTML</p>
12 |    </body>
13 | </html>
```

Si l'intérêt de l'exercice est ici limité, ce genre de méthode nous permet surtout de bien hiérarchiser nos documents XSLT en créant un template associé à un rôle bien précis.

Dans notre cas, nous avons un premier template dont le rôle est de créer la structure de la page HTML tandis que le second template permet d'en écrire le contenu. La relecture et la modification du document XSLT sont ainsi facilitées.

Définition : deuxième partie

Je vous avais promis que l'étude de la fonction `<xsl:call-template />` nous permettrait de mettre en application nos connaissances sur les variables, chose que nous n'avons pas encore faite. Il est donc temps de passer aux choses sérieuses!;)

A l'appel d'un template, il est possible de lui faire passer des paramètres que nous utiliserons comme des variables dans le template appelé. Ces paramètres sont tout simplement des informations que l'on souhaite transmettre au template appelé. Ainsi, il sera par exemple possible de lui faire passer le nom d'une personne afin de l'afficher. Les paramètres s'utilisent exactement comme les variables à ceci près qu'ils se déclarent grâce à la fonction `<xsl:with-param />`. Tout comme la fonction `<xsl:variable />` accepte deux attributs :

- **Name** : c'est le nom que l'on souhaite donner à l'attribut, ce nom va nous permettre de l'identifier de manière unique.
- **Select** : cet attribut optionnel accepte soit une expression XPath qui permet de sélectionner la valeur que l'on souhaite stocker dans notre variable, soit une chaîne de caractères.

```
1 | <xsl:with-param name="nom_du_parametre" select="expression
    XPath|chaîne de caractères" />
```

Voyons alors comment faire passer les paramètres à un template appelé :

```
1 | <xsl:call-template name="nom_du_template">
2 |   <xsl:with-param name="parametre_un">valeur 1</xsl:with-
    param>
3 |   <xsl:with-param name="parametre_deux">valeur 1</xsl:with-
    param>
4 | </xsl:call-template>
```

Comme vous pouvez le constater, il convient simplement d'encapsuler les fonctions `<xsl:with-param />` dans la fonction `<xsl:call-template />`. Nous devons encapsuler autant de fonctions `<xsl:with-param />` que nous voulons de paramètres.

Ce n'est pas tout ! Nous devons également déclarer les paramètres au niveau du template appelé. La déclaration des paramètres attendus par un template ne se fait plus avec la fonction `<xsl:with-param />`, mais la fonction `<xsl:param />` qui accepte deux attributs :

- **Name** : c'est le nom du paramètre que l'on attend.
- **Select** : cet attribut optionnel accepte soit une expression XPath, soit une chaîne de caractères et permet de définir une valeur par défaut au paramètre.

```
1 | <xsl:param name="nom_du_parametre" select="expression XPath|cha
    îne de caractères" />
```

Ainsi, si j'appelle un template de la manière suivante :

```
1 <xsl:call-template name="nom_du_template">
2   <xsl:with-param name="parametre_un">valeur 1</xsl:with-
3     param>
4   <xsl:with-param name="parametre_deux">valeur 1</xsl:with-
5     param>
6 </xsl:call-template>
```

le template appellé bien contenir les 2 paramètres attendus :

```
1 <xsl:template name="nom_du_template">
2   <xsl:param name="parametre_un" />
3   <xsl:param name="parametre_deux" />
4 </xsl:call-template>
```

Comme pour les variables, pour pouvoir utiliser un paramètre et récupérer la valeur qu'il contient, il convient de précéder son nom par le symbole « \$ ».

Je vous propose de terminer ce chapitre par un exemple. Pour cet exemple, nous allons exploiter le document XML suivant :

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <?xml-stylesheet type="text/xsl" href="transformation.xsl" ?>
3 <repertoire>
4   <!-- John DOE -->
5   <personne sexe="masculin">
6     <nom>DOE</nom>
7     <prenom>John</prenom>
8   </personne>
9
10  <!-- Marie POPPINS -->
11  <personne sexe="feminin">
12    <nom>POPPINS</nom>
13    <prenom>Marie</prenom>
14  </personne>
15
16  <!-- Batte MAN -->
17  <personne sexe="masculin">
18    <nom>MAN</nom>
19    <prenom>Batte</prenom>
20  </personne>
21 </repertoire>
```

Nous allons écrire un document XSLT contenant deux templates :

- Le premier template capturera la racine du document XML et construira la structure du document HTML à produire. Il bouclera également sur chaque personne du document XML et appellera un second template en lui donnant paramètre le nom de chaque personne.
- Le second template accueillera 2 paramètres. Le premier sera le nom de la personne tandis que le deuxième aura pour valeur par défaut la chaîne de caractères « nom de la personne ». Le but de ce second template sera d'afficher le nom de la personne.

Voici à quoi ressemble notre document XSLT :

```

1  <xsl:template match="/">
2      <html>
3          <head>
4              <title>Test de la fonction call-template</title>
5          </head>
6          <body>
7              <xsl:for-each select="repertoire/personne">
8                  <xsl:call-template name="afficherNom">
9                      <xsl:with-param name="nomFamille" select="
10                         nom" />
11                  </xsl:call-template>
12              </xsl:for-each>
13          </body>
14      </html>
15  </xsl:template>
16
16 <xsl:template name="afficherNom">
17     <xsl:param name="nomFamille" />
18     <xsl:param name="constante">nom de la personne</xsl:param>
19
20     <p><xsl:value-of select="$constante"/>:<xsl:value-of select
21         ="$nomFamille"/></p>
22 </xsl:template>
```

Le document HTML produit est alors le suivant :

```

1  <!DOCTYPE html
2      PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
3          html4/strict.dtd">
4
5      <html>
6          <head>
7              <meta http-equiv="Content-Type" content="text/html;
8                  charset=UTF-8">
9              <title>Test de la fonction call-template</title>
10         </head>
11         <body>
12             <p>nom de la personne:DOE</p>
13             <p>nom de la personne:POPPINS</p>
14             <p>nom de la personne:MAN</p>
15         </body>
16     </html>
```

D'autres fonctions

La fonction element

Définition

La fonction `<xsl:element />` est une fonction que nous utiliserons principalement pour transformer un document XML en un nouveau document XML. En effet, comme son nom le laisse deviner, cette fonction permet de créer un élément au sens XML, c'est-à-dire une **balise XML**.

Cette fonction accepte plusieurs attributs. Dans le cadre de ce cours, deux nous intéressent :

- L'attribut « **name** », obligatoire, permet de définir le nom de l'élément XML que l'on souhaite créer. Ce nom peut-être une chaîne de caractères ou encore la valeur d'une variable.
- L'attribut « **namespace** », facultatif quant à lui, permet de préfixer l'élément à créer d'un espace de noms.

La syntaxe de la fonction `<xsl:element />` est donc la suivante :

```
1 | <xsl:element name="nom" namespace="espace_de_nom" />
```

A noter : pour définir une valeur à l'élément, il convient tout simplement de la placer entre la balise ouvrante et fermante de la fonction :

```
1 | <xsl:element name="nom" namespace="espace_de_nom">valeur de l'élément </xsl:element>
```

Exemple

Afin d'illustrer l'utilisation de cette nouvelle fonction, je vous propose de voir immédiatement un exemple. Nous allons donc écrire un template dont le rôle est de créer un élément XML ayant pour nom « **couleur** » et pour valeur « **rouge** ».



Dans cet exemple, je ne vais détailler ici que le template, mais dans le document XSLT complet, il convient de décrire le document à produire comme un document XML.

```
1 | <xsl:template match="/">
2 |   <xsl:element name="couleur">rouge</xsl:element>
3 | </xsl:template>
```

Le document XML produit est alors le suivant :

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <couleur>rouge</couleur>
```

La fonction attribute

Définition

La fonction `<xsl:attribute />` est une fonction grâce à laquelle il est possible de créer des attributs à des éléments XML ou HTML.

Dans le cadre de ce tutoriel, nous allons nous intéresser au seul attribut obligatoire pour cette fonction. Il s'agit de l'attribut « **name** » qui permet de définir le nom de l'attribut que l'on souhaite créer.

La syntaxe de la fonction `<xsl:attribute />` est donc la suivante :

```
1 | <xsl:attribute name="nom" />
```

Comme pour la fonction `<xsl:element />`, pour définir la valeur, il convient de la placer entre la balise ouvrante et fermante de la fonction :

```
1 | <xsl:attribute name="nom">valeur de l'élément </xsl:element>
```

Exemple

Pour illustrer la fonction `<xsl:attribute />`, je vous propose de compléter notre exemple précédent. Ainsi, nous allons toujours produire un document XML qui grâce à une transformation XSLT contiendra un élément ayant pour nom « **couleur** » et pour valeur « **rouge** ». La nouveauté est que notre template devra également ajouter l'attribut « **primaire** » à l'élément « **couleur** » afin de préciser s'il s'agit d'une couleur primaire ou non.

```
1 | <xsl:template match="/">
2 |   <xsl:element name="couleur">
3 |     <xsl:attribute name="primaire">oui</xsl:attribute>
4 |     rouge
5 |   </xsl:element>
6 | </xsl:template>
```

Le document XML produit est alors le suivant :

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <couleur primaire="oui">rouge</couleur>
```

La fonction comment

Définition

La fonction `<xsl:comment />` est une fonction qui permet de créer des commentaires dans les documents XML ou HTML à produire. Cette fonction ne prend aucun attribut. Pour définir sa valeur, il suffit simplement de la placer entre la balise ouvrante et fermante de la fonction :

```
1 | <xsl:comment>mon commentaire</xsl:comment>
```

Exemple

Cette fonction est relativement simple d'utilisation par rapport à d'autres fonctions que nous avons vu tout au long de ce tutoriel. Cependant, je vous propose tout de même d'illustrer l'utilisation de cette fonction `<xsl:comment />` par un exemple très simple.

Nous allons donc compléter le template créé précédemment afin d'y ajouter un commentaire ayant pour valeur « **ma couleur** » :

```
1 | <xsl:template match="/">
2 |   <xsl:comment>ma couleur</xsl:comment>
3 |   <xsl:element name="couleur">
4 |     <xsl:attribute name="primaire">oui</xsl:attribute>
5 |     rouge
6 |   </xsl:element>
7 | </xsl:template>
```

Le document XML produit est alors le suivant :

```
1 | <?xml version="1.0" encoding="utf-8"?>
2 | <!--ma couleur-->
3 | <couleur primaire="oui">rouge</couleur>
```

La fonction processing-instruction

Définition

La fonction `<xsl:processing-instruction />`, malgré son nom barbare, permet de créer ce qu'on appelle une **instruction de traitement** dans le document que l'on souhaite produire. Une instruction de traitement est une de ces fameuses balises XML qui début par le caractère « **?** ».

C'est notamment ce qu'on utilise dans les documents XML pour référencer un fichier CSS ou une transformation XSLT :

```
1 | <?xml-stylesheet type="text/xsl" href="transformation.xsl"?>
```

Cette fonction accepte un seul attribut qui est l'attribut « **name** » et qui permet de définir le nom de l'instruction. Le reste des éléments contenus dans l'instruction comme par exemple « **type** » et « **href** » doivent être placés entre la balise ouvrante et fermante de la fonction :

```
1 | <xsl:processing-instruction name="nom_instruction">reste de l'
    instruction</xsl:processing-instruction>
```

Exemple

Afin d'illustrer l'utilisation de cette fonction, je vous propose de continuer à travailler avec le document XML que nous produisons depuis le début de ce chapitre. Ainsi, dans ce document XML, nous allons ajouter une nouvelle référence, un document XSLT fictif portant le nom « **transformation.xsl** » :

```

1 | <xsl:template match="/">
2 |   <xsl:processing-instruction name="xmlstylesheet" type="text/xsl" href="transformation.xsl"></xsl:processing-
|     instruction>
3 |   <xsl:comment>ma couleur</xsl:comment>
4 |   <xsl:element name="couleur">
5 |     <xsl:attribute name="primaire">oui</xsl:attribute>
6 |     rouge
7 |   </xsl:element>
8 | </xsl:template>
```

Le document XML produit est alors le suivant :

```

1 | <?xml version="1.0" encoding="utf-8"?>
2 | <?xmlstylesheet type="text/xsl" href="transformation.xsl"?>
3 | <!--ma couleur-->
4 | <couleur primaire="oui">rouge</couleur>
```

La fonction copy-of

Définition

La fonction **<xsl:copy-of />**, permet de copier tel quel un nœud et ses enfants dans le document que l'on souhaite produire. Cette fonction accepte uniquement l'attribut « **select** » qui permet de sélectionner le nœud à copier en passant par une expression XPath :

```
1 | <xsl:copy-of select="expression XPath" />
```

Exemple

Afin d'illustrer l'utilisation de cette fonction, nous allons transformer le document XML suivant :

```

1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <repertoire>
3 |   <personne sexe="masculin">
4 |     <nom>DOE</nom>
5 |     <prenom>John</prenom>
6 |   </personne>
7 |
8 |   <personne sexe="feminin">
```

```
9      <nom>POPPINS</nom>
10     <prenom>Marie</prenom>
11   </personne>
12
13   <personne sexe="mASCULIN">
14     <nom>BATTE</nom>
15     <prenom>Man</prenom>
16   </personne>
17 </repertoire>
```

Dans la transformation suivante, l'objectif sera de produire un document XML qui ne contient que les personnes de sexe féminin. Voici le template à utiliser :

```
1 <xsl:template match="/">
2   <xsl:element name="repertoire">
3     <xsl:for-each select="repertoire/personne">
4       <xsl:if test="@sexe = 'feminin'">
5         <xsl:copy-of select=". "/>
6       </xsl:if>
7     </xsl:for-each>
8   </xsl:element>
9 </xsl:template>
```

Le document XML produit est alors le suivant :

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <repertoire>
3   <personne sexe="fEMININ">
4     <nom>POPPINS</nom>
5     <prenom>Marie</prenom>
6   </personne>
7 </repertoire>
```

Un exemple avec EditiX

Pour conclure ce chapitre, je vous propose de voir comment effectuer une **transformation XSLT avec EditiX**.

Création du document XML

Commencez par créer dans EditiX un document XML contenant les informations suivantes :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <repertoire>
3   <personne sexe="mASCULIN">
4     <nom>DOE</nom>
5     <prenom>John</prenom>
```

```

6   </personne>
7
8   <personne sexe="feminin">
9     <nom>POPPINS</nom>
10    <prenom>Marie</prenom>
11  </personne>
12
13  <personne sexe="masculin">
14    <nom>BATTE</nom>
15    <prenom>Man</prenom>
16  </personne>
17 </repertoire>

```

Création du document XSLT

Pour créer un nouveau document, vous pouvez sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier **[Ctrl] + [N]**.

Dans la liste qui s'affiche, sélectionnez **XSL Transformations 1.0 for HTML output**, comme indiqué sur la figure 23.1.

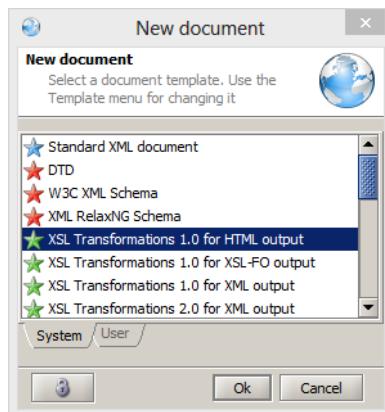


FIGURE 23.1 – Crédit d'un document XSLT

Votre document XSLT n'est normalement pas vierge. Voici ce que vous devriez avoir :

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- New document created with Editix at Tue Nov 05 22:29:40
4      CET 2013 -->
5
6  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999
7      /XSL/Transform">
8
9      <xsl:output method="html"/>

```

```
8     <xsl:template match="/">
9         <html>
10            <body>
11                <xsl:apply-templates />
12            </body>
13        </html>
14    </xsl:template>
15
16
17 </xsl:stylesheet>
```

Remplacez le contenu par notre véritable transformation :

```
1 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999
2   /XSL/Transform">
3
4   <xsl:output
5     method="html"
6     encoding="UTF-8"
7     doctype-public="-//W3C//DTD HTML 4.01//EN"
8     doctype-system="http://www.w3.org/TR/html4/strict.dtd"
9     indent="yes" />
10
11  <xsl:template match="/">
12    <html>
13      <head>
14        <title>Mon répertoire téléphonique</title>
15        <link type="text/css" rel="stylesheet" href="
16          style.css"/>
17      </head>
18      <body>
19        <xsl:for-each select="repertoire/personne">
20          <p><xsl:value-of select="nom"/></p>
21        </xsl:for-each>
22      </body>
23    </html>
24  </xsl:template>
25
26 </xsl:stylesheet>
```

Enregistrez ensuite votre document avec le nom **transformation.xsl** au même endroit que votre document XML.

Vérification du document XSLT

Vous pouvez vérifier que votre transformation n'a pas d'erreur de syntaxe en sélectionnant dans la barre de menu **XML** puis **Check this document** ou encore en utilisant le raccourci clavier **[Ctrl] + [K]**.

Vous devriez normalement voir le message de la figure 23.2 s'afficher.



FIGURE 23.2 – Message indiquant que le document XSLT est bien formé

Appliquer une transformation

Nous allons maintenant voir comment appliquer la transformation que nous venons d'écrire à notre document XML. Pour ce faire, il convient de sélectionner dans la barre de menu **XSLT/Query** puis **Transform a document with this XSLT...** ou encore en cliquant sur l'icône visible à la figure 23.3.



FIGURE 23.3 – Transformer un document

Une fenêtre apparaît alors, nous permettant de choisir :

- La transformation à appliquer.
- Le document XML à transformer.
- Le nom du document qui sera produit à l'issu de la transformation.
- Le comportement souhaité à la fin de l'opération.

Complétez alors les informations comme à la figure 23.4.

Lorsque tout est correctement paramétré, cliquez sur le bouton « **Ok** » pour appliquer la transformation. Une fois celle-ci terminée et en cas de succès, le message visible à la figure 23.5 doit normalement apparaître à l'écran.

Vous pouvez maintenant vérifier que le document HTML produit contient bien le résultat attendu. À noter : un aperçu du résultat de la transformation est disponible dans le module « **XSLT Preview Result** » situé en bas de votre écran lorsque votre document XSLT est ouvert (voir figure 23.6).

En résumé

- Il est possible de déclarer des variables dans un **document XSLT** grâce à la balise `<xsl:variable />`.
- Pour utiliser une variable, il suffit de précéder son nom par le symbole « `$` ».
- Il est possible de faire passer des informations d'un template à un autre sous la forme de **paramètres**.

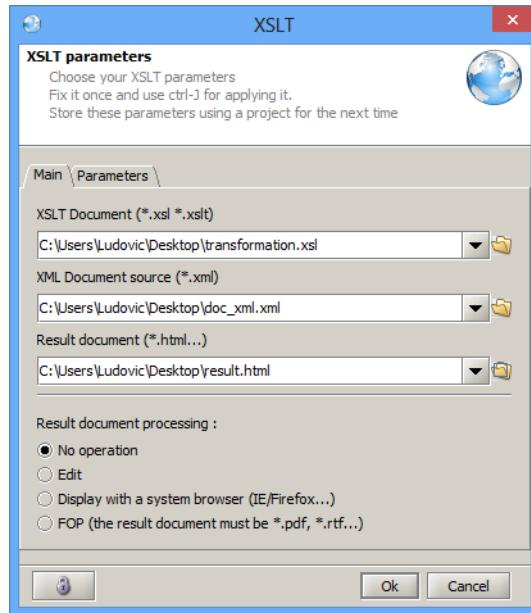


FIGURE 23.4 – Fenêtre d'exécution d'une transformation XSLT

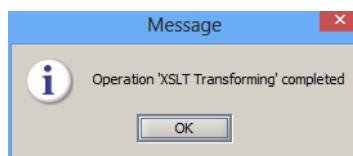


FIGURE 23.5 – Message indiquant que la transformation XSLT a été appliquée



FIGURE 23.6 – Résultat de la transformation XSLT

Chapitre 24

TP : des transformations XSLT d'un répertoire

Difficulté : 

Il est temps de réellement mettre en pratique vos connaissances sur les **transformations XSLT** dans un TP.



Le sujet

L'énoncé

Dans ce TP, le travail demandé consiste à transformer un document XML comportant un répertoire téléphonique en un document HTML dont le rendu final doit être celui visible à la figure 24.1.



FIGURE 24.1 – Rendu du TP

Comme vous pouvez le constater sur la capture d'écran ci-dessus, il vous faudra également styliser le document HTML à l'aide d'un peu de CSS en respectant les règles suivantes :

- Le nom et le prénom d'une personne sont des titres de catégories 1.
 - L'adresse d'une personne doit être écrite en italique.
 - Les entêtes des listes des numéros de téléphone et des adresses e-mails sont des titres de catégories 2.
 - Les listes sont des vraies listes HTML.
 - La couleur de fond est bleue pour un homme et rose pour une femme.

- Les numéros de téléphone et les adresses e-mails apparaissent uniquement si la personne en possède.

Le document XML

Le document XML à transformer est le suivant :

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <adresse>
8       <numero>7</numero>
9       <voie type="impassage">impasse du chemin</voie>
10      <codePostal>75015</codePostal>
11      <ville>PARIS</ville>
12      <pays>FRANCE</pays>
13    </adresse>
14    <telephones>
15      <telephone type="fixe">01 02 03 04 05</telephone>
16      <telephone type="portable">06 07 08 09 10</
17        telephone>
18    </telephones>
19    <emails>
20      <email type="personnel">john.doe@wanadoo.fr</email>
21      <email type="professionnel">john.doe@societe.com</
22        email>
23    </emails>
24  </personne>
25
26  <!-- Marie POPPINS -->
27  <personne sexe="feminin">
28    <nom>POPPINS</nom>
29    <prenom>Marie</prenom>
30    <adresse>
31      <numero>28</numero>
32      <voie type="avenue">avenue de la république</voie>
33      <codePostal>13005</codePostal>
34      <ville>MARSEILLE</ville>
35      <pays>FRANCE</pays>
36    </adresse>
37    <telephones>
38      <telephone type="professionnel">04 05 06 07 08</
39        telephone>
40    </telephones>
41    <emails>
42      <email type="professionnel">contact@poppins.fr</
43        email>

```

```
40      </emails>
41  </personne>
42
43      <!-- Batte MAN -->
44  <personne sexe="mASCULIN">
45      <nom>MAN</nom>
46      <prenom>Batte</prenom>
47      <adresse>
48          <numero>24</numero>
49          <voie type="avenue">impasse des héros</voie>
50          <codePostal>11004</codePostal>
51          <ville>GOTHAM CITY</ville>
52          <pays>USA</pays>
53      </adresse>
54      <telephones>
55          <telephone type="professionnel">01 03 05 07 09</
56              telephone>
57      </telephones>
58  </personne>
59 </repertoire>
```

▷ Copier ce code
Code web : [534435](#)

Une solution

Comme à chaque fois, je vous fais part de ma solution, que vous trouverez grâce au code web suivant.

▷ Voir ma solution
Code web : [934455](#)

Cinquième partie

Annexes

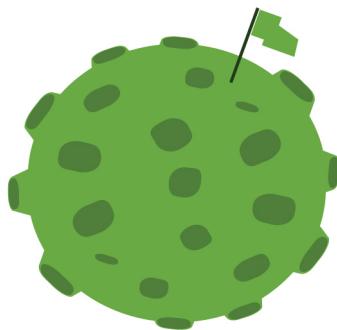
Chapitre 25

Les espaces de noms

Difficulté : 

Comme vous le constatez, cette cinquième partie est une annexe. Son objectif est multiple, à savoir revenir sur des notions qui reviennent régulièrement dans le cours et proposer quelques chapitres « bonus » pour aller toujours plus loin dans la manipulation de vos documents XML.

Le premier chapitre de cette annexe revient sur la notion d'**espace de noms**, notion fortement utilisé en XML et dans les différentes technologies abordées au cours de ce tutoriel comme par exemple les **Schémas XML** ou encore les **transformations XSLT**.



Définition

Définition d'un espace de noms

Lorsque l'on écrit un document XML, on utilise ce que l'on appelle un **vocabulaire**. Par exemple, dans les différents TP, nous avons travaillé avec des répertoires téléphoniques dans lesquels chaque personne possède :

- Une identité (un nom et un prénom).
- Une adresse.
- Des numéros de téléphone.
- Des adresses e-mail.
- Etc.

A travers cette description, nous avons avons défini le vocabulaire d'une personne.

Dans notre cas, ce vocabulaire n'est pas forcément réutilisable en l'état. Pourtant, comme vous vous en êtes certainement aperçu, en informatique, on aime bien réutiliser ce que l'on a déjà effectué et ne pas toujours repartir à zéro.

Il existe plusieurs vocabulaires qui ont fait leurs preuves et qui ont été mis à disposition des développeurs afin qu'ils puissent être réutilisés. Nous y reviendrons un peu plus tard, mais nous pouvons déjà en citer quelques uns.

- Le vocabulaire permettant de décrire une **page xHTML**.
- Le vocabulaire permettant de décrire un **Schéma XML**.
- Le vocabulaire permettant de décrire des documents techniques.
- Etc.

Identifier un espace de noms

Un espace de noms est identifié par une **URI** (Uniform Resource Identifier) qui permet de l'identifier de manière unique. Bien que l'on distingue 2 types d'URI, à savoir les **URL** (Uniform Resource Locator) et les **URN** (Uniform Resource Name), dans la majorité des cas, c'est une URL qui est utilisée.

Pour rappel, une URL permet d'identifier de manière unique une ressource, dans notre cas, un vocabulaire, sur un réseau.

Voyons quelques exemples d'URL permettant d'identifier des vocabulaires et donc des espaces de noms sur le réseau Internet :

- xhtml : <http://www.w3.org/1999/xhtml>
- Schéma XML : <http://www.w3.org/2001/XMLSchema>
- DocBook : <http://docbook.org/ns/docbook>

Utilisation d'un espace de noms

Les espaces de noms par défaut

La **déclaration d'un espace de noms par défaut** se fait dans le premier élément qui utilise le vocabulaire, grâce au mot clef **xmlns** comme **XML namespace**.

```
1 | xmlns="mon_uri"
```

Illustrons alors la déclaration et l'utilisation d'un espace de noms par défaut à travers l'exemple d'un document xHTMLM :

```
1 | <html xmlns="http://www.w3.org/1999/xhtml">
2 |   <head>
3 |     <title>Titre du document</title>
4 |   </head>
5 |   <body>
6 |     <p>
7 |       
8 |
9 |       <a href="mon_lien">Mon super lien !</a>
10 |
11 |     </p>
12 |   </body>
13 | </html>
```

Tous les éléments utilisés dans ce document XML comme `<head />`, `<title />`, `<body />`, etc., font partie du vocabulaire d'une page xHTMLM. C'est grâce à la déclaration de l'espace de noms dans l'élément `<html />` que nous pouvons utiliser les différents éléments du vocabulaire tout en respectant les règles qui régissent leurs imbrications les uns par rapport aux autres.

Les espaces de noms avec préfixe

Utiliser un **espace de noms par défaut** a certaines limites. En effet, il est par exemple impossible d'utiliser au sein d'un même document 2 espaces de noms par défaut qui auraient un mot de vocabulaire en commun. On obtiendrait alors une ambiguïté.

Tout problème a bien évidemment une solution. Ainsi, pour contourner cette limite, nous allons utiliser des préfixes avec nos espaces de noms.

Tout comme pour un espace de noms par défaut, la déclaration d'un **espace de noms avec préfixe** se fait dans le premier élément qui utilise le vocabulaire, grâce au mot clef **xmlns :prefixe**.

```
1 | xmlns:prefixe="mon_uri"
```

Lorsqu'un espace de noms est déclaré avec un préfixe, tous les éléments qui appartiennent au vocabulaire, et donc à l'espace de noms, doivent être précédés par ce préfixe :

```
1 | <prefixe:element />
```

Afin d'illustrer cette nouvelle notion, reprenons la page xHTML que nous avons écrit plus haut et utilisons cette fois-ci un préfixe :

```
1 <http:html xmlns:http="http://www.w3.org/1999/xhtml">
2   <http:head>
3     <http:title>Titre du document</http:title>
4   </http:head>
5   <http:body>
6     <http:p>
7       <http:img src="mon_image.png" alt="ma super image"
8         />
9       <http:br/>
10      <http:a href="mon_lien">Mon super lien !</http:a>
11    </http:p>
12  </http:body>
13 </http:html>
```

La portée d'un espace de noms

Pour clore ce chapitre, il me semble intéressant de revenir sur la notion de **portée d'un espace de noms**. En effet, suivant la façon dont il est déclaré, un espace de noms n'est pas accessible partout dans un document, il n'est donc pas possible d'utiliser tous les mots de vocabulaire partout dans un document XML.

La règle qui régit la portée d'un espace de noms est assez simple : *un espace de noms est utilisable tant que l'élément qui le déclare n'est pas refermé*.

Une fois de plus, je vous propose d'illustrer cette notion par un exemple :

```
1 <!-- il est possible d'utiliser l'espace de noms http -->
2 <http:html xmlns:http="http://www.w3.org/1999/xhtml">
3   <http:head>
4     <http:title>Titre du document</http:title>
5   </http:head>
6   <http:body>
7     <http:p>
8       <!-- il est possible d'utiliser l'espace de noms ml
9         -->
10      <ml:math xmlns:ml="http://www.w3.org/1998/Math/
11        MathML">
12        <ml:matrix>
13          <ml:matrixrow>
14            <ml:cn>0</ml:cn>
15            <ml:cn>1</ml:cn>
16            <ml:cn>0</ml:cn>
17          </ml:matrixrow>
18          <ml:matrixrow>
19            <ml:cn>0</ml:cn>
              <ml:cn>0</ml:cn>
              <ml:cn>1</ml:cn>
```

```
20          </ml:matrixrow>
21      </ml:matrix>
22  </ml:math>
23  <!-- il n'est plus possible d'utiliser l'espace de
     noms ml -->
24  </http:p>
25  </http:body>
26 </http:html>
27 <!-- il n'est plus possible d'utiliser l'espace de noms http --
    >
```

Quelques espaces de noms utilisés régulièrement

En conclusion de ce chapitre, je vous propose de revenir sur quelques espaces de noms connus, régulièrement utilisés par les développeurs. Bien évidemment cette liste n'est pas exhaustive, il en existe un très grand nombre !

DocBook

DocBook permet de décrire des documents techniques comme des livres, des articles, etc.

Cet espace de noms est identifié par l'URI <http://docbook.org/ns/docbook>.

MathML

MathML est une spécification du W3C qui permet d'afficher éléments mathématiques divers et variés comme des additions, des soustractions, des matrices, etc.

Cet espace de noms est identifié par l'URI <http://www.w3.org/1998/Math/MathML>.

Schéma XML

Il s'agit d'une spécification du W3C qui permet de décrire des Schémas XML.

Cet espace de noms est identifié par l'URI <http://www.w3.org/2001/XMLSchema>.

SVG

SVG pour Scalable Vector Graphics est une spécification du W3C qui permet de décrire des images vectorielles.

Cet espace de noms est identifié par l'URI <http://www.w3.org/2000/svg>.

XLink

Il s'agit d'une spécification du W3C permettant de créer des liens entre plusieurs fichiers XML.

Cet espace de noms est identifié par l'URI <http://www.w3.org/1999/xlink>.

XSLT

XSLT pour **eXtensible Stylesheet Language Transformations** est une spécification du W3C qui permet de décrire des transformations à appliquer un document XML.

Cet espace de noms est identifié par l'URI <http://www.w3.org/1999/XSL/Transform>.

En résumé

- Un **espace de noms** est identifié de manière unique par une **URI**.
- Un **espace de noms** se déclare dans le premier élément qui utilise son vocabulaire.
- Un **espace de noms** se déclare grâce à l'attribut **xmlns**.

Chapitre 26

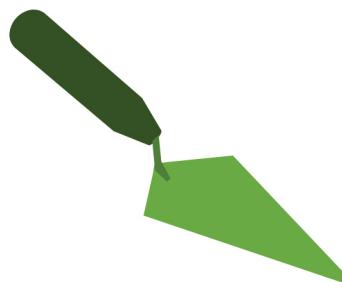
Mettez en forme vos documents XML avec CSS

Difficulté : 

Ce chapitre a pour objectif de vous montrer qu'il est possible d'appliquer un style CSS directement aux informations d'un document XML afin de le mettre en forme lors de son affichage dans un navigateur internet. Finalement, puisque la pratique reste le meilleur moyen d'apprendre, un petit TP vous sera proposé !



Si ce chapitre explique comment appliquer une feuille de style CSS à un document XML, je ne reviendrai pas en détail sur chaque propriété du CSS existante. En effet, ce langage nécessite à lui seul l'écriture d'un tutoriel dont la taille serait conséquente. Le but est ici de découvrir ce qu'il est possible de faire avec du CSS et du XML.



Ecrire un document CSS

Qu'est-ce que le CSS ?

Bien que je sois certain que la majorité d'entre vous connaissent déjà à peu près ce qu'est le **CSS**, je vous propose de revenir rapidement sur ce langage.

Le **CSS** ou **Cascading Style Sheet** de son nom complet, est un langage informatique à part entière permettant de **styliser les documents HTML et XML**. Avec le HTML, il s'agit du langage qui permet de mettre en forme et styliser les sites Internet.

Nous allons donc voir comment grâce au **CSS**, il est possible de **styliser un document XML**. Par styliser, j'entends par exemple :

- Ecrire certains éléments en gras.
- Souligner.
- Surligner.
- Ecrire certains éléments en couleur.
- Etc.

Où écrire le CSS ?

Vous commencez à le comprendre, en informatique, on aime bien séparer les différentes technologies dans différents fichiers. Le **CSS** ne fait pas exception à la règle. Ainsi, nos lignes de CSS seront écrites dans un fichier portant l'extension de fichier **.css**.

Référencer le fichier CSS

Afin de pouvoir appliquer un **style CSS** à nos **documents XML**, il convient de lier les différents fichiers entre eux.

Cette liaison se fait dans le document XML entre le **prologue** et la **racine**, grâce à la balise suivante :

```
1 | <?xml-stylesheet href="style.css" type="text/css" ?>
```

Prenons par exemple le document XML suivant :

```
1 | <?xml version = "1.0" encoding="UTF-8"?>
2 | <?xml-stylesheet href="personne.css" type="text/css" ?>
3 | <personne>
4 |   <nom>NORRIS</nom>
5 |   <prenom>Chuck</prenom>
6 | </personne>
```

Dans cet exemple, l'affichage du document XML sera stylisé grâce au document CSS **personne.css**.

Syntaxe du CSS

Comme tous les langages, le **CSS** dispose d'une **syntaxe** qui lui est propre. L'idée du langage est de sélectionner un ou plusieurs éléments d'un document afin d'y appliquer un certain nombre de propriétés.

Dans ce chapitre, je ne vais pas revenir sur l'ensemble des propriétés qui existent en CSS. Cependant, je vous encourage à lire le mémo écrit par Mathieu Nebra dans son cours sur HTML5 et CSS3.

▷ [Lire le memento](#)
[Code web : 586304](#)

Sélectionner une balise

En **CSS**, pour sélectionner un élément particulier d'un document XML, on utilise la syntaxe suivante :

```
1 | balise {  
2 |     propriété1 : valeur;  
3 |     propriété2: valeur;  
4 | }
```

Illustrons cette première règle grâce à un exemple. Soit le document XML suivant représentant une liste de personnes :

```
1 | <?xml version="1.0" encoding="UTF-8"?>  
2 | <?xml-stylesheet href="personnes.css" type="text/css" ?>  
3 | <personnes>  
4 |     <personne>  
5 |         <nom>NORRIS</nom>  
6 |         <prenom>Chuck</prenom>  
7 |     </personne>  
8 |  
9 |     <personne>  
10 |         <nom>DUPONT</nom>  
11 |         <prenom>Marie</prenom>  
12 |     </personne>  
13 | </personnes>
```

Si je souhaite par exemple afficher les prénoms des différentes personnes en rouge, gras et italique. Écrivons alors le contenu du fichier **personnes.css** :

```
1 | nom {  
2 |     color:red;  
3 |     font-weight:bold;  
4 |     font-style:italic;  
5 | }
```

Si vous ouvrez votre document XML dans un navigateur web, vous devriez alors avoir l'affichage indiqué sur la figure 26.1.

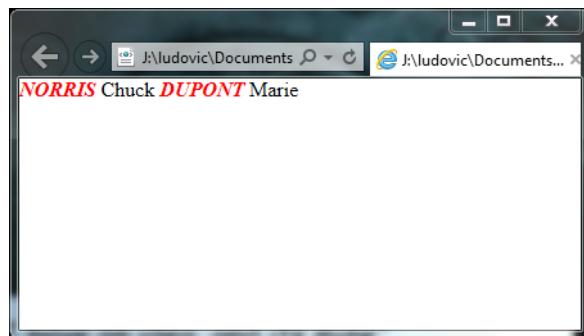


FIGURE 26.1 – Affichage du document XML stylisé dans un navigateur web

Comme on le souhaitait, les noms des personnes sont bien écrits en rouge, gras et italique.

Sélectionner une balise particulière

Allons un petit peu plus loin et considérons le document XML suivant, représentant encore une liste de personnes :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <personnes>
3   <personne sexe="masculin">
4     <nom>NORRIS</nom>
5     <prenom>Chuck</prenom>
6   </personne>
7
8   <personne sexe="feminin">
9     <nom>DUPONT</nom>
10    <prenom>Marie</prenom>
11  </personne>
12 </personnes>
```

Comment faire si je souhaite, par exemple, afficher le nom des hommes en bleu et celui des femmes en roses ? Pour arriver à ce résultat, il convient de sélectionner une personne en fonction de l'attribut `sexe`.

Pour **sélectionner une balise** particulière en fonction de la valeur d'un attribut en CSS, on utilise la syntaxe suivante :

```
1 | balise[attribut="valeur"] {
2 |   propriété1 : valeur;
3 |   propriété2: valeur;
4 | }
```

Tentons alors d'appliquer le style suivant à notre document XML :

```
1 | personne[sexe="masculin"] { color:blue; }
```

```
2 | personne[sexe="feminin"] { color:pink; }
```

Si l'on affiche le document XML dans un navigateur web, on obtient le résultat affiché en figure 26.2.

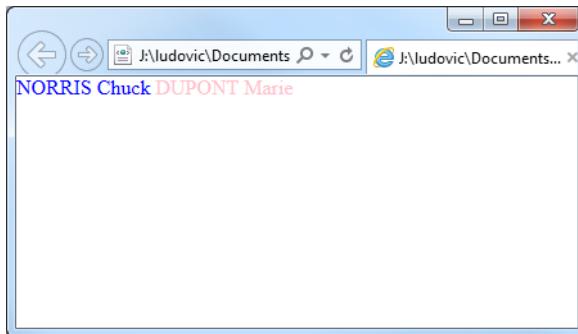


FIGURE 26.2 – Affichage du document XML stylisé dans un navigateur web

On s'approche du résultat souhaité, mais ce n'est pas encore tout à fait ça. En effet, actuellement, les noms et prénoms des personnes sont colorées, or ce que l'on souhaite nous, c'est qu'uniquement des noms des personnes soient colorés.

Il nous suffit de légèrement modifier notre feuille de style :

```
1 | personne[sexe="masculin"] nom { color:blue; }
2 | personne[sexe="feminin"] nom { color:pink; }
```

Comme vous pouvez le constater, on a ajouté l'élément **nom** au niveau de notre sélection. En français, on pourrait traduire ces lignes de CSS par les 2 phrases suivantes :

- Ecrit en bleu le nom des personnes de sexe masculin.
- Ecrit en rose le nom des personnes de sexe féminin.

Finalement, si vous affichez le document XML dans un navigateur web, vous devriez avoir le résultat affiché en figure 26.3, correspondant bien à nos attentes.

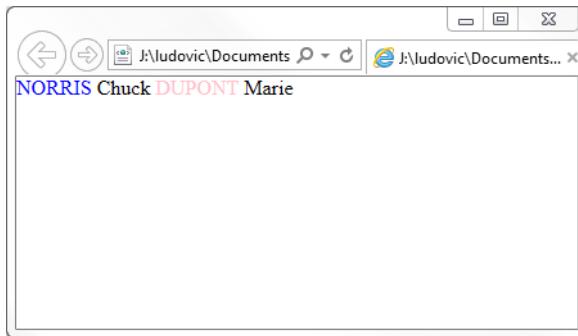


FIGURE 26.3 – Affichage du document XML stylisé dans un navigateur web

Un exemple avec EditiX

Comme pour chaque technologie que nous voyons ensemble, je vous propose de voir comment procéder grâce au logiciel **EditiX**.

Création du document XML

La **création du document XML** n'a rien de bien compliqué puisque nous l'avons déjà vu ensemble à plusieurs reprise.

Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil à la page 21.

Je vous propose de reprendre pour exemple notre liste de personnes :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet href="personnes.css" type="text/css" ?>
3 <personnes>
4   <personne sexe="masculin">
5     <nom>NORRIS</nom>
6     <prenom>Chuck</prenom>
7   </personne>
8
9   <personne sexe="feminin">
10    <nom>DUPONT</nom>
11    <prenom>Marie</prenom>
12  </personne>
13 </personnes>
```

Si vous essayez de lancer la vérification du document, vous devriez normalement avoir ce message (voir la figure 26.4).

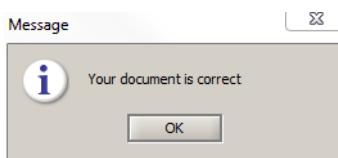


FIGURE 26.4 – Message indiquant que le document XML est bien formé

Création du document CSS

Pour créer un nouveau document, vous pouvez sélectionner dans la barre de menu **File** puis **New** ou utiliser le raccourci clavier **Ctrl** + **N**.

Dans la liste qui s'affiche, sélectionnez **CSS**, comme indiqué sur la figure 26.5.

Votre document CSS n'est normalement pas vierge. Voici ce que vous devriez avoir :

```
1 | /* Generated with EditiX at Sat May 11 19:46:07 CEST 2013 */
```

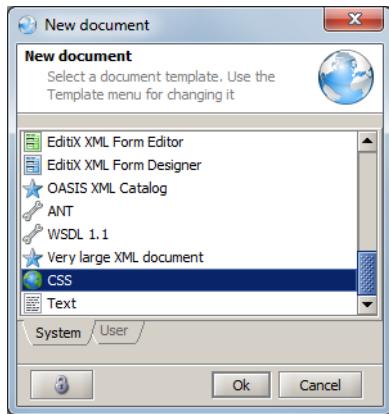


FIGURE 26.5 – Création d'une feuille de style CSS

Replacez le contenu par notre véritable CSS :

```
1 | personne[sexe="masculin"] nom { color:blue; }
2 | personne[sexe="feminin"] nom { color:pink; }
```

Enregistrez ensuite votre document avec le nom **personnes.css** au même endroit que votre document XML.

Vérification de fichier de style

Vous pouvez vérifier que votre fichier CSS n'a pas d'erreur de syntaxe en cliquant sur l'icône adéquat, en sélectionnant dans la barre de menu **XML** puis **Check this CSS** ou encore en utilisant le raccourci clavier **Ctrl** + **K**.



La possibilité de vérifier qu'un fichier CSS ne comporte aucune erreur de syntaxe n'est pas offerte dans la version gratuite d>Editix. Si vous souhaitez utiliser cette fonctionnalité, il vous faudra passer sur la version payante.

Vous devriez normalement avoir un message indiquant que votre CSS est correct, comme l'illustre la figure 26.6.



FIGURE 26.6 – Message indiquant que le document CSS est correct

Lorsque tout est correct, ouvrez votre fichier XML dans un navigateur web pour ob-

server le résultat.

TP : mise en forme d'un répertoire

Le but de ce TP est de **créer un fichier CSS** afin de mettre en forme un répertoire téléphonique se présentant sous la forme d'un document XML.

Le document XML

Voici le document XML à mettre en forme :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <personnes>
3     <personne sexe="mASCULIN">
4         <identite>
5             <nom>NORRIS</nom>
6             <prenom>Chuck</prenom>
7         </identite>
8         <telephones>
9             <telephone type="fixe">01 02 03 04 05</telephone>
10            <telephone type="portable">06 07 08 09 10</
11                telephone>
12            </telephones>
13        </personne>
14
15        <personne sexe="fEMININ">
16            <identite>
17                <nom>DUPONT</nom>
18                <prenom>Marie</prenom>
19            </identite>
20            <telephones>
21                <telephone type="bureau">04 05 06 07 08</telephone>
22            </telephones>
23        </personne>
24
25        <personne sexe="mASCULIN">
26            <identite>
27                <nom>PAUL</nom>
28                <prenom>Bernard</prenom>
29            </identite>
30            <telephones>
31                <telephone type="portable">07 08 09 10 11</
32                    telephone>
33            </telephones>
34        </personne>
35    </personnes>
```

La mise en forme

Voici à la figure 26.7 une capture d'écran de la mise en forme que vous devez reproduire.

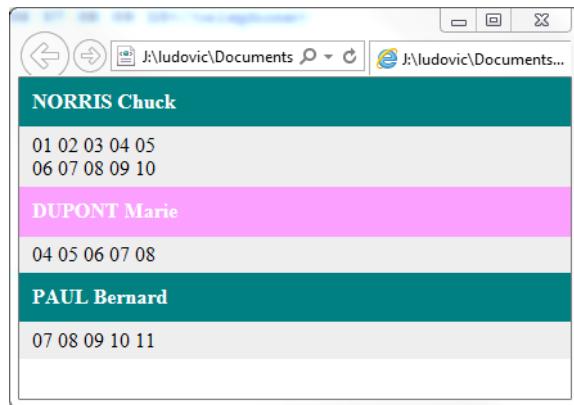


FIGURE 26.7 – Mise en page à reproduire

Comme vous pouvez le constater, l'identité d'une personne de sexe masculin est sur fond bleu tandis que celle d'une personne de sexe féminin est sur fond rose. Tous les numéros de téléphone sont sur fond gris.

Une solution

Pour ne pas changer, voici un exemple de solution.

Le fichier XML

Tout d'abord, voyons le document XML avec le fichier de style CSS référencé :

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet href="personnes.css" type="text/css" ?>
3 <personnes>
4   <personne sexe="masculin">
5     <identite>
6       <nom>NORRIS</nom>
7       <prenom>Chuck</prenom>
8     </identite>
9     <telephones>
10       <telephone type="fixe">01 02 03 04 05</telephone>
11       <telephone type="portable">06 07 08 09 10</
12         telephone>
13     </telephones>
14   </personne>
15
<personne sexe="feminin">
```

```
16     <identite>
17         <nom> DUPONT </nom>
18         <prenom> Marie </prenom>
19     </identite>
20     <telephones>
21         <telephone type="bureau"> 04 05 06 07 08 </telephone>
22     </telephones>
23 </personne>
24
25 <personne sexe="masculin">
26     <identite>
27         <nom> PAUL </nom>
28         <prenom> Bernard </prenom>
29     </identite>
30     <telephones>
31         <telephone type="portable"> 07 08 09 10 11 </
32             telephone>
33     </telephones>
34 </personne>
35 </personnes>
```

Le fichier CSS

Revenons maintenant sur le fichier CSS :

```
1 personne {
2     display: block;
3 }
4
5 identite {
6     display: block;
7     padding: 10px;
8     font-weight: bold;
9     color: white;
10 }
11
12 personne[sexe="masculin"] identite {
13     background-color: #008080;
14 }
15
16 personne[sexe="feminin"] identite {
17     background-color: #FBAOFE;
18 }
19
20 telephones {
21     display: block;
22     background-color: #EEE;
23     padding: 5px 10px;
24 }
```

```
25 | telephone {  
26 |     display:block;  
27 | }  
28 | }
```

En résumé

- il est possible d’appliquer directement une feuille de style CSS à un document XML.

