

Documents et outils XML

Pierre Nerzic - pierre.nerzic@univ-rennes1.fr

février-mars 2020

Abstract

Il s'agit des transparents du cours mis sous une forme plus facilement imprimable et lisible. Ces documents ne sont pas totalement libres de droits. Ce sont des supports de cours mis à votre disposition pour vos études sous la licence *Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International*.



Version du 26/05/2020 à 09:15

Table des matières

1	Concepts de base	13
1.1	Introduction	13
1.1.1	Pourquoi ce cours ?	13
1.1.2	Exemple de fichier XML	13
1.1.3	Applications de XML	14
1.1.4	XML en tant que format de fichier	14
1.1.5	Bases de données XML	14
1.1.6	Échange de données entre clients et serveur	15
1.2	Historique	15
1.2.1	Origine de XML	15
1.2.2	Chronologie	15
1.2.3	Exemple de document GML	15
1.3	Structure d'un document XML	16
1.3.1	Arborescence d'éléments	16
1.3.2	Exemple complet	16

1.3.3	Représentation graphique	16
1.3.4	Explications	17
1.3.5	Vocabulaire	17
1.3.6	Vocabulaire (suite)	17
1.4	Détails du format XML	18
1.4.1	Prologue XML	18
1.4.2	Norme Unicode	18
1.4.3	Commentaires XML	18
1.4.4	Attention aux -- dans les commentaires	19
1.4.5	Options après le prologue	19
1.4.6	Éléments	19
1.4.7	Choses interdites	20
1.4.8	Choses permises	20
1.4.9	Noms des éléments	20
1.4.10	Espaces de nommage	21
1.4.11	Remarques	21
1.4.12	Évolution des normes	21
1.4.13	Définition d'un espace de nommage	22
1.4.14	Exemple revu	22
1.4.15	Namespace par défaut	22
1.4.16	Attributs	23
1.4.17	Attributs dans l'arbre du document	23
1.4.18	Entités	23
1.4.19	Exemple	24
1.4.20	Entités définies dans le document	24
1.4.21	Exemple d'entité interne	24
1.4.22	Entités externes	25
1.4.23	Texte	25
1.4.24	Arbre correspondant	25
1.4.25	Sections CDATA	25
1.4.26	Fusion des CDATA et textes	26
1.5	Modélisation	26
1.5.1	Modélisation d'un TE	26
1.5.2	Propriétés dans les attributs ou des sous-éléments ?	27

1.5.3	Attributs ou sous-éléments ? (suite)	27
1.5.4	Attributs ou contenu ? (fin)	27
1.5.5	Associations	28
2	Validation d'un document XML	29
2.1	Validité d'un document	29
2.1.1	Introduction	29
2.1.2	Processus de validation	29
2.2	Document Type Definitions (DTD)	30
2.2.1	Présentation	30
2.2.2	Intégration d'une DTD	30
2.2.3	Outils de validation d'un document avec DTD	30
2.2.4	Contenu d'une DTD	31
2.2.5	Racine du document	31
2.2.6	Définition d'un élément	31
2.2.7	Exemple de contenus	31
2.2.8	Définition de sous-éléments	32
2.2.9	Contenus alternatifs	32
2.2.10	Définition des attributs	33
2.2.11	Types d'attributs	33
2.2.12	Définition d'entités (rappel et précisions)	33
2.2.13	Entités paramètres	34
2.3	XML Schemas	34
2.3.1	Présentation	34
2.3.2	Association entre un document et un schéma local	34
2.3.3	Association entre un document et un schéma public	35
2.3.4	Principes généraux des Schémas XML	35
2.3.5	Structure générale d'un schéma	36
2.3.6	Remarque importante	36
2.3.7	Définition d'éléments	36
2.3.8	Définition de types de données	37
2.3.9	Types de données (suite)	37
2.3.10	Types de données (suite)	37
2.3.11	Restrictions sur les types	38

2.3.12	Définition de restrictions	38
2.3.13	Restriction communes à tous les types	38
2.3.14	Restrictions communes (suite)	39
2.3.15	Restrictions sur les dates et nombres	39
2.3.16	Types à alternatives	39
2.3.17	Types à alternatives (suite)	39
2.3.18	Exemple de type à alternatives	40
2.3.19	Données de type liste	40
2.3.20	Exemple de liste	41
2.3.21	Contenu d'éléments	41
2.3.22	Type complexe	41
2.3.23	Contenu d'un type complexe	42
2.3.24	Exemple de séquence	42
2.3.25	Exemple de choix	42
2.3.26	Imbrication de structures	43
2.3.27	Nombre de répétitions	43
2.3.28	Définition d'attributs	44
2.3.29	Cas spéciaux	44
2.3.30	Élément vide sans attribut	44
2.3.31	Élément vide avec attribut	44
2.3.32	Élément texte sans attribut	45
2.3.33	Élément texte avec attribut	45
2.3.34	Éléments enfants sans attribut	46
2.3.35	Éléments enfants avec attribut	46
2.3.36	Éléments enfants avec texte mélangé	47

3 RelaxNG et XPath 48

3.1	RelaxNG	48
3.1.1	Présentation	48
3.1.2	Exemple de document à valider	48
3.1.3	DTD du document	48
3.1.4	XML Schema du document	49
3.1.5	Feuille RelaxNG	49
3.1.6	Deux syntaxes pour RelaxNG	50

3.1.7	Principes de RelaxNG, syntaxe compacte	50
3.1.8	Ordonnancement des éléments enfants	51
3.1.9	Exemple de successions dans le contenu	51
3.1.10	Types XSD et contraintes	51
3.1.11	Alternatives	52
3.1.12	Types liste	52
3.1.13	Syntaxe compacte nommée	52
3.1.14	Exemple	53
3.2	XPath	53
3.2.1	Présentation	53
3.2.2	Parcours d'arbre	54
3.2.3	Principe général	54
3.2.4	Réponses multiples	55
3.2.5	Évaluation d'une expression XPath	55
3.2.6	XPath en JavaScript	55
3.2.7	Structure d'une expression XPath simple	56
3.2.8	Exemples	56
3.2.9	Attributs des éléments	56
3.2.10	Autres étapes d'un chemin	56
3.2.11	Remarque sur l'opérateur d'alternative	57
3.2.12	Conditions sur les étapes	57
3.2.13	Syntaxe des conditions	57
3.2.14	Opérateurs de comparaison	58
3.2.15	Fonctions XPath	58
3.2.16	Fonctions XPath (suite)	58
3.2.17	Fonctions XPath (suite)	59
3.2.18	Fonctions XPath (suite)	59
3.2.19	Retour sur les composants d'un chemin	59
3.2.20	Axes	59
3.2.21	Algorithme de XPath	60
3.2.22	Exemple	60
3.2.23	Exemple (suite)	60
3.2.24	Exemple (suite et fin)	61
3.2.25	Axes	61
3.2.26	Exemples de chemins avec axes	61

4	Transformation d'un document	62
4.1	Feuilles de styles CSS	62
4.1.1	Feuille CSS pour un document XML	62
4.1.2	Exemple de document XML	62
4.1.3	Exemple de feuille de style CSS	63
4.2	XSLT	63
4.2.1	Présentation	63
4.2.2	Exemple de feuille de style	63
4.2.3	Entête d'une feuille XSLT	64
4.2.4	Outil de transformation <code>xsltproc</code>	64
4.2.5	Principe général	64
4.2.6	Exemple de traitement	65
4.2.7	Patrons	65
4.2.8	Exemple de patron	66
4.2.9	Créer des éléments et des attributs	66
4.2.10	<code>value-of</code> et <code>copy-of</code>	67
4.2.11	Fonctions utiles	67
4.3	Structures de contrôle XSLT	67
4.3.1	Variables	67
4.3.2	Conditionnelles	68
4.3.3	Exemple de contenu conditionnel	68
4.3.4	Comment faire un <i>else</i> ?	68
4.3.5	Remarques sur les tests	69
4.3.6	Boucle sur les nœuds enfant	69
4.3.7	Exemple de patron de type <i>boucle</i>	69
4.3.8	Exemple de patron (suite)	70
4.3.9	Tri des itérations	70
4.3.10	Remarque sur les boucles	70
4.3.11	XSL 2.0	71
4.3.12	Index en XSL	71
4.3.13	Indentifiants d'éléments	72
4.3.14	Groupement de Steve Muench	72
4.3.15	Traitement d'un document complexe	72
4.3.16	Exemple de patrons imbriqués	72

5	XQuery et les bases de données XML	74
5.1	XQuery	74
5.1.1	Présentation	74
5.1.2	Exemple initial	74
5.1.3	Traitement d'une feuille XQuery	75
5.1.4	Bases de XQuery	75
5.1.5	Scripts XQuery	75
5.1.6	Génération d'éléments XML	75
5.1.7	Affectation de variables	76
5.1.8	Affectations multiples	76
5.1.9	Conditionnelles	77
5.1.10	Conditionnelles (suite)	77
5.1.11	Boucles	77
5.1.12	Clause For	78
5.1.13	Clause For sur des attributs	78
5.1.14	Clause Let	78
5.1.15	Clause Where	79
5.1.16	Clause Order by	79
5.1.17	Boucles imbriquées	79
5.2	Bases de données XML	80
5.2.1	Présentation	80
5.2.2	Principe général d'un SGBD XML	80
5.2.3	Utilisation de BaseX	80
5.2.4	Interface de BaseX	81
5.2.5	Interface graphique (suite)	81
5.2.6	Création d'une base de données XML	81
5.2.7	Requête XQuery	82
5.3	XQuery pour modifier la base XML	82
5.3.1	Requêtes en modification	82
5.3.2	Insertion d'éléments	82
5.3.3	Insertion sur plusieurs éléments	83
5.3.4	Insertion d'attributs	83
5.3.5	Suppression d'éléments ou d'attributs	84
5.3.6	Remplacement d'éléments ou d'attributs	84

5.3.7	Remplacement d'éléments ou d'attributs	84
5.3.8	Autres actions	84
6	API W3C DOM	85
6.1	Principes	85
6.1.1	Présentation	85
6.1.2	Principe généraux de l'API DOM	85
6.1.3	Bibliothèques	86
6.2	Document DOM en mode création	86
6.2.1	Création d'un Document	86
6.2.2	Compléments	86
6.2.3	Création d'éléments	87
6.2.4	Insertion avant un autre élément	87
6.2.5	Création d'un arbre d'éléments	87
6.2.6	Ajout d'attributs aux éléments	88
6.2.7	Espaces de nommage	88
6.2.8	Ajout de textes et de CDATA	88
6.2.9	Ajout de commentaires et autres	89
6.2.10	Enregistrement dans un fichier	89
6.3	Document DOM en mode lecture	90
6.3.1	Traitement du document	90
6.3.2	Ouverture d'un fichier	90
6.3.3	Classe Node	90
6.3.4	Modification d'un document	90
6.3.5	Prologue du document	91
6.3.6	Élément racine	91
6.3.7	Espaces de nommages	91
6.3.8	Attributs d'un Element	92
6.3.9	Nœuds enfants d'un élément	92
6.3.10	Voisinage d'un nœud	92
6.3.11	Parcours des nœuds enfants (méthode 1)	93
6.3.12	Parcours des nœuds enfants (méthode 2)	93
6.3.13	Parcours des nœuds enfants (méthode 3)	93
6.3.14	Parcours des nœuds enfants (méthode 4)	93

6.3.15	Traitement d'un nœud	94
6.3.16	Traitement d'un élément	94
6.3.17	Traitement d'éléments	95
6.3.18	Contenu d'un nœud texte	95
6.4	Sérialisation d'objets Java	95
6.4.1	Principes	95
6.4.2	Exemple	96
6.4.3	Récursivité	96
6.4.4	Sérialisation manuelle	96
6.4.5	Exemple de sérialisation	97
6.4.6	Cas des collections	97
6.4.7	Principes de désérialisation manuelle	98
6.5	Sérialisation avec des outils	99
6.5.1	Présentation	99
6.6	Sérialisation avec l'API XStream	99
6.6.1	Présentation	99
6.6.2	Sérialisation avec XStream	100
6.6.3	Désérialisation avec XStream	100
6.6.4	Objet <code>XStream</code>	100
6.6.5	Ajustement de la sortie XML	101
6.7	Sérialisation avec l'API Simple	102
6.7.1	Présentation	102
6.7.2	Annotations	102
6.7.3	Exemple	102
6.7.4	Sérialisation	103
6.7.5	Désérialisation	103
6.7.6	Ajustement de la sortie XML	103
6.8	API DOM dans d'autres langages	104
6.8.1	Résumé	104
6.8.2	Création d'un document XML en JavaScript	104
6.8.3	Script de création d'un document	104
6.8.4	Création d'éléments	105
6.8.5	Affichage du résultat	105
6.8.6	Parcours d'un fichier XML	105

6.8.7	Traitement de la réponse HTTP	106
6.9	Validation en JAVA	106
6.9.1	Présentation	106
6.9.2	Validation par un schéma	106
7	API SAX	108
7.1	Simple API for XML	108
7.1.1	Présentation	108
7.1.2	Principes de SAX	108
7.1.3	Fonctionnement de SAX	108
7.1.4	Interface <code>ContentHandler</code>	109
7.1.5	Type <code>Attributes</code>	109
7.1.6	Interface <code>ContentHandler</code> (suite)	109
7.1.7	Texte, CDATA et entités	110
7.2	Programmation d'un analyseur	110
7.2.1	Implémentation d'un <code>ContentHandler</code>	110
7.2.2	Lancement de l'analyse	111
7.2.3	Gestion des erreurs	111
7.3	Traitement d'un document XML	111
7.3.1	Aucune visibilité globale	111
7.3.2	Mémoriser les informations au passage	112
7.3.3	Automate à états	112
7.3.4	Programmation d'un automate à états	113
7.3.5	Machine de Mealy	113
7.3.6	Application à l'analyse SAX	113
7.3.7	Traitements des transitions	113
7.3.8	Traitements des transitions (suite)	114
7.3.9	Concaténation de tous les textes	114
7.3.10	Traitements des transitions (fin)	115
7.4	API <code>XMLWriter</code> de PHP	115
7.4.1	Présentation	115
7.4.2	Ouverture du flux de sortie	116
7.4.3	Écriture d'éléments	116

8	XML dans le SGBD PostgreSQL	117
8.1	XML dans un SGBD	117
8.1.1	Présentation	117
8.1.2	Stockage de données XML	117
8.1.3	Texte vers XML	117
8.1.4	Suffixe ::XML ou mot clé XML	118
8.1.5	XML vers Texte	118
8.1.6	PostgreSQL et XPath	118
8.1.7	Génération de XML à partir de données normales	119
8.1.8	Génération du XML d'un n-uplet	119
8.1.9	Fonction XMLELEMENT	119
8.1.10	Contenu d'un XMLELEMENT	120
8.1.11	Génération du XML d'un n-uplet (suite)	120
8.1.12	Regroupement de fragments XML	120
8.1.13	Regroupement de fragments XML (suite)	121
8.1.14	Concaténation d'éléments	121
8.1.15	Un contenu plus facile à écrire	122
8.1.16	Entête du document	122
8.1.17	Racine du document	122
8.1.18	Fournir une DTD	122
8.2	PHP, PostgreSQL et XML	123
8.2.1	Présentation	123
8.2.2	Utilisation de l'API XMLWriter	123
8.2.3	Ouverture de la base	123
8.2.4	Création d'un écrivain XML	124
8.2.5	Création d'un écrivain XML	124
8.2.6	Terminaison	124
8.2.7	Encodage par le SGBD	124
8.2.8	Requête SQL	125
8.2.9	Reste du script PHP	125
8.2.10	Comparaisons	125
8.3	Autres formats de données internet	125
8.3.1	Alternatives au XML	125
8.3.2	JSON	126

8.3.3	Schéma de JSON	126
8.3.4	Suite du schéma	126
8.3.5	Outils de validation	127
8.3.6	Sérialisation JSON	127
8.3.7	Dé-sérialisation JSON	127
8.3.8	YAML	128

Semaine 1

Concepts de base

XML = Extensible Markup Language

C'est un langage permettant de représenter et structurer des informations à l'aide de balises que chacun peut définir et employer comme il le veut.

```
texte ... <BALISE> ... texte </BALISE> ...
```

Le cours de cette semaine présente les concepts de base :

- Applications typiques,
- Historique,
- Structure d'un document XML.

1.1. Introduction

1.1.1. Pourquoi ce cours ?

L'objectif est de connaître les principaux formats d'échanges de données entre logiciels : enregistrement de fichiers, communication entre serveur et clients, etc.

Les formats XML, JSON, YAML sont intéressants car ce sont des textes lisibles et structurés. Ils sont également accompagnés d'outils de vérification, transformation et interrogation. Ils sont directement liés à des structures de données dans la plupart des langages de programmation : un objet Java par exemple peut être lu ou écrit dans un fichier JSON ou XML.

Il faut noter que JavaScript semble prendre de plus en plus d'importance et il est fortement lié au format JSON.

Inversement Java semble traverser des turbulences avec des retards importants dans les nouvelles versions et des ruptures de compatibilité. Plusieurs API XML pour Java ne se compilent pas avec Java 9.

XML est le format le plus ancien, le plus polyvalent et le plus complet. C'est lui que nous étudierons principalement. Connaître XML permet d'utiliser les autres sans aucune difficulté.

1.1.2. Exemple de fichier XML

Un fichier XML représente des informations structurées :



```
<?xml version="1.0" encoding="utf-8"?>
<!-- itinéraire fictif -->
<itineraire>
  <etape distance="0km">départ</etape>
  <etape distance="13km">tourner à droite</etape>
  <etape distance="22km">arrivée</etape>
</itineraire>
```

Cet exemple modélise un itinéraire composé d'étapes.

XML permet de choisir la représentation des données sans aucune contrainte. Il faut seulement que ça soit homogène, régulier.

1.1.3. Applications de XML

Le format XML est au cœur de nombreux processus actuels :

- format d'enregistrement de nombreuses applications,
- échange de données entre serveurs et clients,
- outils et langages de programmation,
- bases de données XML natives.

Il faut comprendre que XML définit la syntaxe des documents, mais les applications définissent les balises, leur signification et ce qu'elles doivent contenir. Des API existent dans tous les langages pour lire et écrire des documents XML.

Voici une liste des applications un peu plus détaillée.

1.1.4. XML en tant que format de fichier

XML est la norme d'enregistrement de nombreux logiciels parmi lesquels on peut citer :

- Bureautique
 - LibreOffice : format [OpenDocument](#)
 - Publication de livres et documentations : [DocBook](#)
- Graphismes
 - Dessin vectoriel avec Inkscape : format [SVG](#)
 - Équations mathématiques : format [MathML](#)
- Programmation
 - Interfaces graphiques : Android, [XUL](#) (mais le futur [WebExtensions](#) est tourné vers JS et JSON)
 - Construction/compilation de projets : [Ant](#), [Maven](#)
- Divers
 - Itinéraires GPS : format [GPX](#)

1.1.5. Bases de données XML

XML permet aussi de représenter des données complexes.

- [Web sémantique](#) : c'est un projet qui vise à faire en sorte que toutes les connaissances présentes plus ou moins explicitement dans les pages web puissent devenir accessibles par des mécanismes

de recherche unifiés. Pour cela, il faudrait employer des marqueurs portant du sens en plus de marqueurs de mise en page, par exemple rajouter des balises indiquant que telle information est le prix unitaire d'un article.

L'un des mécanismes du Web sémantique est une base de données appelée [RDF](#). Elle peut être interrogée à l'aide d'un langage de requêtes appelé [SparQL](#).

- Bases de données [XML native](#) : les données sont au format XML et les requêtes sont dans un langage (XQuery) permettant de réaliser l'équivalent de SQL.

1.1.6. Échange de données entre clients et serveur

XML est le format utilisé pour représenter des données volatiles de nombreux protocoles dont :

- Les flux [RSS](#) permettent de résumer les changements survenus sur un site Web. Par exemple, un site d'information émet des messages RSS indiquant les dernières nouvelles.
- Les protocoles [XML-RPC](#) et [SOAP](#) permettent d'exécuter des procédures à distance (*Remote Procedure Call*) : un client demande à un serveur d'exécuter un programme ou une requête de base de donnée puis attend les résultats. Le protocole gère l'encodage de la requête et des résultats, indépendamment des langages de programmation et des systèmes employés.
- Autre exemple : [AJAX](#) permet de mettre à jour dynamiquement les pages web sur le navigateur.

1.2. Historique

1.2.1. Origine de XML

XML est un cousin de HTML. Tous deux sont les successeurs de SGML, lui-même issu de GML. Ce dernier a été conçu par IBM pour dissocier l'apparence des documents textes de leur contenu, en particulier des documentations techniques. Il est facile de changer l'apparence (mise en page, polices, couleurs...) d'un tel document sans en ré-écrire une seule ligne. D'autres langages, comme \LaTeX sont similaires : on écrit le texte sans se soucier de la mise en page. Cela permet également de le traduire dans d'autres langues.

Inversement, les logiciels *wysiwyg* comme Word mélangent mise en page et contenu. Même avec des styles, il reste difficile de remanier l'apparence d'un document sans devoir le ré-écrire partiellement.

XML permet de représenter beaucoup plus que des textes.

1.2.2. Chronologie

1969 : GML (IBM) GML est un langage d'écriture de documents techniques, défini par IBM, destiné à être traité par un logiciel propriétaire de mise en page appelé [SCRIPT/VS](#).

1990 : SGML et HTML SGML est une norme libre très complexe dont HTML est un sous-ensemble très spécialisé. L'une des raisons qui ont conduit à les définir est d'améliorer la pérennité des documentations.

1998 : XML c'est une généralisation de SGML permettant de construire toutes sortes de documents.

1.2.3. Exemple de document GML

Les balises sont notées :TAG. ou :TAG attr=valeur. quand il y a des attributs. Les balises fermantes sont notées :ETAG. Cela ressemble beaucoup à la syntaxe <TAG> et </TAG>. L'une des plus utilisées est :P. pour créer un paragraphe.



```
:H1 id=exemple.Exemple de texte GML
:P.Certaines balises ressemblent un peu à celles de HTML
mais pas toutes. Voici du :HP2.texte en gras:EHP2. et un
exemple:FNREF refid=note1. :
:FN id=note1.Voici une note de bas de page.:EFN.
:XMP.
xmlstarlet c14n document.xml
:EXMP.
```

1.3. Structure d'un document XML

1.3.1. Arborescence d'éléments

Un document XML est composé de plusieurs parties :

- Entête de document précisant la version et l'encodage,
- Des règles optionnelles permettant de vérifier si le document est valide (c'est l'objet des deux prochains cours),
- Un arbre d'*éléments* basé sur un élément appelé *racine*
 - Un *élément* possède un *nom*, des *attributs* et un *contenu*
 - Le contenu d'un élément est :
 - * du texte
 - * d'autres éléments (les éléments enfants).
 - Un élément est marqué par une *balise ouvrante* et une *balise fermante*.
 - * une balise ouvrante est notée `<nom attributs...>`
 - * une balise fermante est notée `</nom>`

1.3.2. Exemple complet

Voici un document XML représentant une personne :



```
<?xml version="1.0" encoding="utf-8"?>
<personne>
  <nom>Nerzic</nom>
  <prenom>Pierre</prenom>
  <profession>
    enseignant
  <lieu>
    IUT
    <ville>Lannion</ville>
  </lieu>
  informatique
</profession>
</personne>
```

1.3.3. Représentation graphique

Voir la figure 1, page 17.

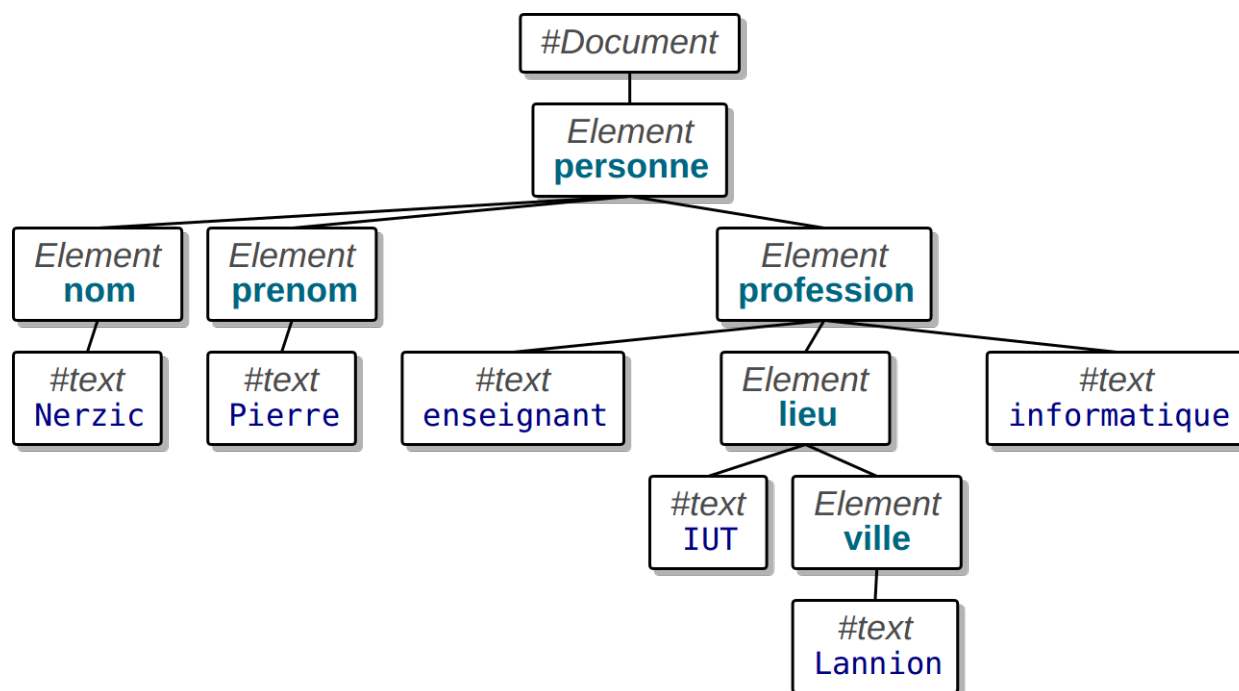


Figure 1: Arbre XML

1.3.4. Explications

Le document XML représente un arbre composé de plusieurs types de nœuds :

nœuds éléments ils sont associés aux balises `<element>`. Ce sont des nœuds qui peuvent avoir des enfants en dessous.

nœuds #text ils représentent le texte situé entre deux balises. Les nœuds texte sont des feuilles dans l'arbre.

Notez que différents textes peuvent être entrelacés avec des éléments. Voir le cas de `<lieu>` dans le contenu de `<profession>`. Il est possible de le faire, mais ce n'est pas forcément souhaitable.

Les autres types de nœuds seront présentés au fur et à mesure.

1.3.5. Vocabulaire

Soit cet arbre XML :

Voir la figure 2, page 18.

1.3.6. Vocabulaire (suite)

Voici comment on désigne les différents nœuds les uns par rapport aux autres :

- `<racine>` est le nœud **parent** du nœud **enfant** (*child*) `<enfant3>`, lui-même parent de `<enfant4>` et `<enfant5>`,
- `<racine>`, `<enfant3>` sont des nœuds **ancêtres** (*ancestors*) de `<enfant4>` et `<enfant5>`,
- `<enfant4>` et `<enfant5>` sont des **descendants** (*descendants*) de `<racine>` et `<enfant3>`,
- `<enfant1>` est un nœud **frère** (*sibling*) de `<enfant2>` et réciproquement.

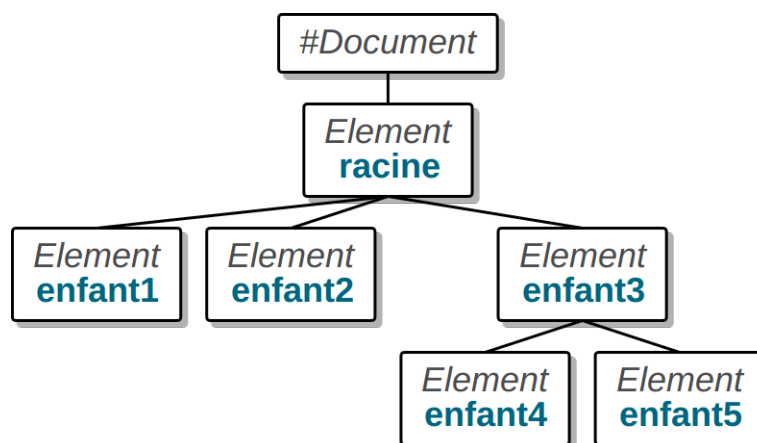


Figure 2: Nœuds parent, enfant et cousins

1.4. Détails du format XML

1.4.1. Prologue XML

La première ligne d'un document XML est appelée *prologue XML*. Elle spécifie la version de la norme XML utilisée (1.0 ou 1.1 qui sont très similaires¹) ainsi que l'encodage :

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

```
<?xml version="1.0" encoding="iso-8859-15"?>
```

1.4.2. Norme Unicode

La norme [Unicode](#) définit un ensemble abstrait de plus de 245000 caractères représentable sur un ordinateur, par exemple é, €, «, ©... Ces caractères doivent être codés sous forme d'octets. C'est là qu'il y a plusieurs normes :

- [ASCII](#) ne représente que les 128 premiers caractères Unicode.
- [ISO 8859-1](#) ou Latin-1 représente 191 caractères de l'alphabet latin n°1 (ouest de l'europe) à l'aide d'un seul octet. Les caractères € et œ n'en font pas partie, pourtant il y a æ.
- [ISO 8859-15](#) est une norme mieux adaptée au français. Elle rajoute € et œ et supprime des caractères peu utiles comme æ.
- [UTF-8](#) représente les caractères à l'aide d'un nombre variable d'octets, de 1 à 4 selon le caractère. Par exemple : A est codé par 0x41, é par 0xC3,0xA9 et € par 0xE2,0x82,0xAC.

1.4.3. Commentaires XML

On peut placer des commentaires à peu près partout dans un document XML. La syntaxe est identique à celle d'un fichier HTML. Un commentaire peut d'étendre sur plusieurs lignes. La seule contrainte est de ne pas pouvoir employer les caractères -- dans le commentaire, même s'ils ne sont pas suivis de >

¹La norme 1.1 est plus claire concernant certains nouveaux caractères unicode qui peuvent spécifier des retours à la ligne et autres caractères de contrôle.

```
<!-- voici un commentaire -->
ceci n'est pas un commentaire
<!--
    voici un autre commentaire
    et ça -- , c'est une erreur
-->
```

1.4.4. Attention aux -- dans les commentaires

En fait, en XML (comme en SGML), `<!--` et `-->` ne sont pas des marques de commentaires comme `/*` et `*/` en C.

- `<!` marque le début d'une instruction de traitement destinée à l'analyseur
- `>` signale la fin de cette instruction de traitement
- `--` inverse le mode « commentaire », entre « hors commentaire » et « dans un commentaire ».


Ainsi, un commentaire `<!--commentaire-->` est analysé ainsi :

1. `<!` début d'une instruction de traitement
2. `--` passage en mode commentaire
3. `commentaire` : ignoré, on peut donc tout y mettre sauf `--`
4. `--` sortie du mode commentaire, ne rien mettre après
5. `>` sortie de l'instruction de traitement

1.4.5. Options après le prologue

Après le prologue, on peut trouver plusieurs parties optionnelles délimitées par `<?...?>` ou `<!...>`. Ce sont des instructions de traitement, des directives pour l'analyseur XML.

Par exemple :

- un *Document Type Definitions* (DTD) qui permet de valider le contenu du document. C'est l'objet du prochain cours.
- une feuille de style, voir cours 4. 

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE personne SYSTEM "personne.dtd">
<?xml-stylesheet href="personne.css" type="text/css"?>
<personne>
    ...
</personne>
```

1.4.6. Éléments

Les éléments définissent la structure du document. Un élément est délimité par :

- une balise ouvrante `<nom attributs...>`
- une balise fermante `</nom>` obligatoire.

Le contenu de l'élément se trouve entre les deux balises. Ce sont des éléments enfants et/ou du texte.

```
<parent>
  texte1
  <enfant>texte2</enfant>
  texte3
</parent>
```

Dans le cas où l'élément n'a pas de contenu (*élément vide*), on peut regrouper ses deux balises en une seule `<nom attributs.../>`

1.4.7. Choses interdites

Les règles d'imbrication XML interdisent différentes configurations qui sont plus ou moins tolérées en HTML :

- plusieurs racines dans le document,
- des éléments non terminés (NB: XML est sensible à la casse),
- des éléments qui se chevauchent.

```
<element1>
  <element2>
    </Element2>
  <element3>
    </element1>
</element3>
```

En XML, cela crée des erreurs « *document mal formé* ».

1.4.8. Choses permises

Parmi les choses permises, le même type d'élément peut se trouver plusieurs fois avec le même parent, avec des contenus identiques ou différents (à vous de définir si ça a du sens) :

```
<element1>
  <element2>texte1</element2>
  <element2>texte2</element2>
  <element2>texte1</element2>
</element1>
```

Il est possible d'interdire cela, cependant ce n'est pas relatif à la syntaxe XML mais à la validation du document par rapport à une spécification, et aussi ce qu'on veut faire avec le document. Voir les DTD et les schémas dans le prochain cours.

1.4.9. Noms des éléments

Les noms des éléments peuvent employer de nombreux caractères Unicode (correspondant au codage déclaré dans le prologue) mais pas les signes de ponctuation.

- L'initiale doit être une lettre ou le signe _


- ensuite, on peut trouver des lettres, des chiffres ou - . _

Le caractère `:` permet de séparer le nom en deux parties : préfixe et *nom local*. Le tout s'appelle *nom qualifié*. Par exemple `iut:departement` est un nom qualifié préfixé par `iut`. Ce préfixe permet de définir un *espace de nommage*.

$$\text{nom qualifié} = \text{préfixe}:\text{nom local}$$

1.4.10. Espaces de nommage

Un espace de nommage (*namespace*) définit une famille de noms afin d'éviter les confusions entre des éléments qui auraient le même nom mais pas le même sens. Cela arrive quand le document XML modélise les informations de plusieurs domaines.

Voici un exemple dans le domaine de la vente de meubles. Le document modélise une table (avec 4 pieds) et aussi un tableau HTML pour afficher ses dimensions. On voit la confusion. 

```
<meuble id="765">
  <table prix="74,99€">acajou</table>
  <table border="1">
    <tr><th>longueur</th><th>largeur</th></tr>
    <tr><td>120cm</td><td>80cm</td></tr>
  </table>
</meuble>
```

1.4.11. Remarques

Le document de l'exemple précédent modélise une situation particulière. On veut que le document contienne à la fois une description en XML et du code HTML. Sans doute, la partie HTML sera extraite et affichée à un utilisateur.

C'est finalement ce qu'on fait du document, son utilisation, ses transformations qui guident sa construction. Nous verrons tout ce qui concerne la transformation d'un document XML en semaine 4.

L'exemple précédent est un peu artificiel puisqu'on a fait exprès de choisir le même nom pour des éléments de nature différente. On verra dans un prochain cours qu'il est pratique d'employer des espaces de nommage quand on mélange au sein du même document XML des instructions de traitement et des données. C'est le cas dans les feuilles de style XSLT.

1.4.12. Évolution des normes

Une autre raison pour employer des espaces de nommage est de permettre l'extension d'une norme qui risque d'évoluer par la suite. Par exemple la norme [GPX](#) qui définit des fichiers de géolocalisation des GPS. Elle définit par exemple la structure d'un point de trace `<trkpt>`. Il a un certain nombre d'attributs comme `lon` et `lat` pour les coordonnées géographiques, et de sous-éléments tels que `<ele>` qui mémorisent l'altitude d'un point.

La société Garmin a rajouté de nouvelles informations pour les sportifs tels que `<hr>` pour mémoriser le rythme cardiaque. Or il y a le risque que dans l'avenir, le format GPX évolue et définisse de son côté un élément `<hr>` avec une autre signification.

Grâce à un espace de nommage spécifique, la société Garmin peut définir ses propres balises `<garmin:hr>` sans risquer de conflit avec la norme GPX générale.

1.4.13. Définition d'un espace de nommage

On doit choisir un [URI](#), par exemple un URL, pour identifier l'espace de nommage. Cet URI n'a pas besoin de correspondre à un véritable contenu car il ne sera pas téléchargé.

Un URI est la généralisation d'un [URL](#) : schéma : [//[user:passwd@]hôte[:port]] [/]chemin[?requête] [#fragment], par exemple `http://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Syntax`.

Les [URN](#) sont au format `urn:NID:NSS`, ex: `urn:iutlan:xmlsem1`

Ensuite on rajoute un attribut spécial à la racine du document :

```
<?xml version="1.0" encoding="utf-8"?>
<racine xmlns:PREFIXE="URI">
  <PREFIXE:balise>...</PREFIXE:balise>
</racine>
```

1.4.14. Exemple revu

Voici l'exemple précédent, avec deux *namespaces*, un URN pour les meubles et un URL pour HTML :



```
<?xml version="1.0" encoding="utf-8"?>
<meuble:meuble id="765"
  xmlns:meuble="urn:iutlan:meubles"
  xmlns:html="http://www.w3.org">
  <meuble:table prix="74,99€">acajou</meuble:table>
  <html:table border="1">
    <html:tr><html:th>longueur</html:th>...</html:tr>
    <html:tr><html:td>120cm</html:td>...</html:tr>
  </html:table>
</meuble:meuble>
```

Notez le préfixe également appliqué à la racine du document.

1.4.15. Namespace par défaut

Lorsque la racine du document définit un attribut `xmlns="URI"`, alors par défaut toutes les balises du document sont placées dans ce namespace, donc pas besoin de mettre un préfixe.

```
<?xml version="1.0" encoding="utf-8"?>
<book xmlns="http://docbook.org/ns/docbook">
```

Ça peut s'appliquer également localement à un élément et ça concerne toute sa descendance :

```
<meuble id="765" xmlns="urn:iutlan:meubles">
  <table prix="74,99€">acajou</table>
  <table border="1" xmlns="http://www.w3.org">
    <tr><th>longueur</th>...</tr>
```

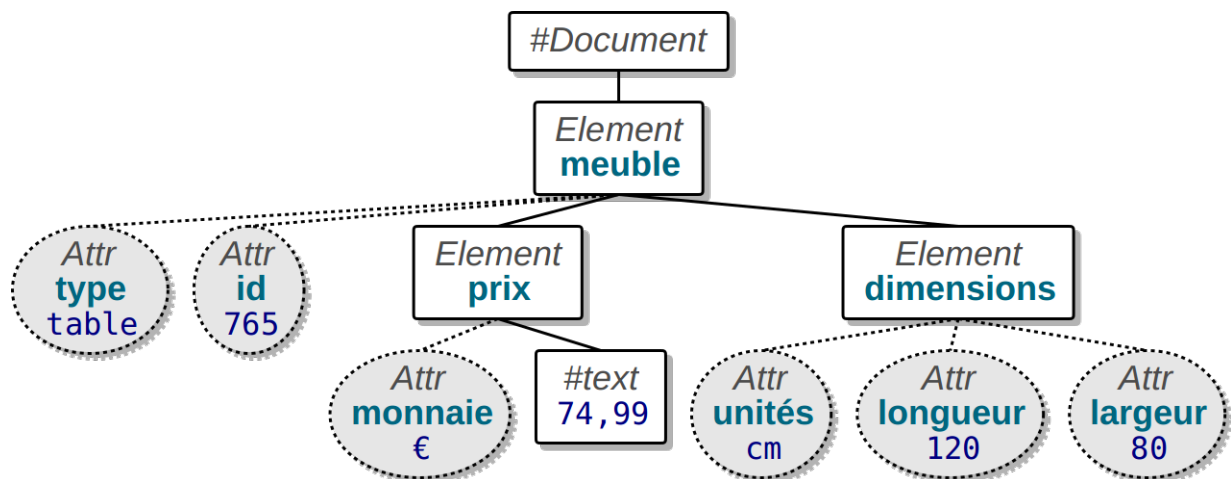



Figure 3: Arbre XML

```
<tr><td>120cm</td>...</tr>
</table>
</meuble>
```

1.4.16. Attributs

Les attributs caractérisent un élément. Ce sont des couples nom="valeur" ou nom='valeur'. Ils sont placés dans la balise ouvrante. 

```
<?xml version="1.0" encoding="utf-8"?>
<meuble id="765" type='table'>
  <prix monnaie='€'>74,99</prix>
  <dimensions unites="cm" longueur="120" largeur="80"/>
  <description langue='fr'>Belle petite table</description>
</meuble>
```

Remarques :

- Il n'y a pas d'ordre entre les attributs d'un élément,
- Un attribut ne peut être présent qu'une fois.

1.4.17. Attributs dans l'arbre du document

Les attributs sont vus comme des nœuds dans l'arbre sous-jacent. Voici l'arbre d'une partie de l'exemple précédent :

1.4.18. Entités

Certains caractères sont interdits dans le contenu des éléments. Comme il est interdit de les employer, XML propose une représentation appelée *référence d'entité* ou *entité* pour simplifier. On retrouve le même concept en HTML.

Caractère	Entité
<	<
>	>
&	&
,	'
"	"

1.4.19. Exemple

Voici un exemple d'emploi d'entités :


```
<?xml version="1.0" encoding="utf-8"?>
<achat-si type-groupement="&amp;&amp;">
  <test>prix &lt; 50.00</test>
  <test>description &lt;&gt; <str value="&quot;&quot;"/></test>
</achat-si>
```

- Cela entraîne une erreur si on remplace ces entités par le caractère correspondant,
- Ces entités sont automatiquement remplacées par les caractères lorsqu'on traite le fichier.

NB: ce contenu bizarre vise uniquement à illustrer l'emploi d'entités.

1.4.20. Entités définies dans le document

Il est possible de créer ses propres entités. Dans ce cas, elles peuvent représenter beaucoup plus qu'un caractère, toute une chaîne, voire même tout un arbre XML.

Voici comment faire. Il faut définir un nœud DOCTYPE appelé *Document Type Definitions* (DTD) concernant la racine du document. Normalement, on y rajoute les règles indiquant la structure du document XML, mais ici, il n'y a que les entités. 

```
<!DOCTYPE racine [
  <!ENTITY nom "texte à mettre à la place">
]>
```

L'entité &nom; est appelée *entité interne*.

Il faut aussi rajouter standalone="no" dans le prologue du document.

1.4.21. Exemple d'entité interne



```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE personne [
  <!ENTITY adresse "IUT de Lannion - Dept Informatique">
  <!ENTITY cours "Cours XML">
]>
<personne>
  <lieu>&adresse;</lieu>
  <role>&cours;</role>
</personne>
```


Pour afficher le résultat, on peut l'ouvrir dans Firefox ou utiliser `xmllint` :

```
xmllint --noent exemple.xml
```


1.4.22. Entités externes

Il s'agit d'une entité qui représente tout un document XML contenu dans un autre fichier. Voici comment on la définit : 

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE racine [
    <!ENTITY voiture1 SYSTEM "voiture1.xml">
]>
<garage>
    <vente>&voiture1;</vente>
</garage>
```

- Le mot clé **SYSTEM** commande la lecture d'un URL, ici, c'est un fichier local
- Le contenu du document `voiture1.xml` est inséré à la place de l'entité `&voiture1`;

1.4.23. Texte

Les textes font partie du contenu des éléments et sont vus comme des nœuds enfants. Il faut bien comprendre que la totalité du texte situé entre les balises, y compris les espaces et retour à la ligne font partie du texte. 


```
<?xml version="1.0" encoding="utf-8"?>
<racine>
    texte1
    <enfant>texte2</enfant>
    texte3
</racine>
```

1.4.24. Arbre correspondant

Voici l'arbre correspondant à l'exemple précédent. Notez les retours à la ligne et espaces présents dans les textes sauf `texte2`.

Voir la figure 4, page 26.

1.4.25. Sections CDATA

Lorsqu'on veut écrire du texte brut à ne pas analyser en tant qu'XML, on emploie une section CDATA : 

```
<?xml version="1.0" encoding="utf-8"?>
<fichier>
    <nom>exemple1.xml</nom>
    <md5><![CDATA[19573b741c0c5c8190a83430967bfa58]]></md5>
</fichier>
```

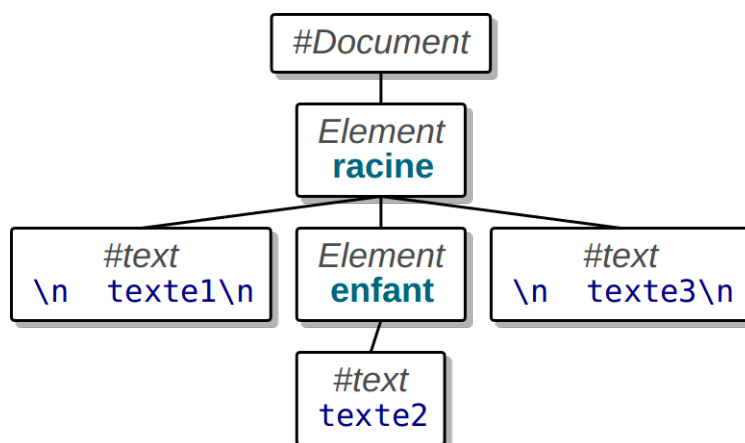


Figure 4: Arbre XML

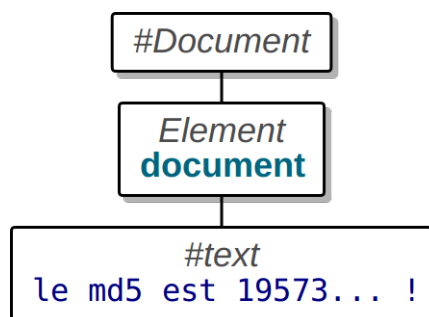


Figure 5: Arbre XML

La partie entre `<![CDATA[` et `]]>` est ignorée par les analyseurs XML, on peut mettre n'importe quoi sauf `]]>`. Ces données sont considérées comme du texte par les analyseurs.

Utiliser `xmllint --nocdata document.xml` pour le voir sans les marqueurs.

1.4.26. Fusion des CDATA et textes

S'il y a des textes avant ou après une section CDATA, ils sont tous fusionnés avec elle.

```
<?xml version="1.0" encoding="utf-8"?>
<document>le md5 est <![CDATA[19573...]]> !</document>
```

1.5. Modélisation

1.5.1. Modélisation d'un TE

XML sert à représenter des informations. Regardons comment cela se passe pour un modèle entité-association, en commençant par la modélisation d'un TE, puis d'un TA en XML.

Pour un TE, il est simple de les représenter à l'aide d'éléments.

Voici par exemple une liste de voitures, chacune ayant un identifiant, une marque et une couleur :

```
<?xml version="1.0" encoding="utf-8"?>
<voitures>
  <voiture id="125" marque="Renault" couleur="grise"/>
  <voiture id="982" marque="Peugeot" couleur="noire"/>
</voitures>
```

1.5.2. Propriétés dans les attributs ou des sous-éléments ?

En ce qui concerne les propriétés de l'entité, on peut les placer :

- dans les attributs des éléments :

```
<voiture id="125" marque="Renault" couleur="grise"/>
```

- en tant que sous-éléments :

```
<voiture>
  <id>125</id>
  <marque>Renault</marque>
  <couleur>grise</couleur>
</voiture>
```

- ou employer un mélange des deux.

Ces choix dépendent principalement des traitements qu'on souhaite effectuer sur le document ; voir le cours de la semaine 4.

1.5.3. Attributs ou sous-éléments ? (suite)

Dans la réflexion pour choisir de représenter une information sous forme d'attributs ou de sous-éléments, il faut prendre en compte ces critères :

- Les attributs ne peuvent pas contenir plusieurs valeurs, tandis qu'on peut avoir plusieurs sous-éléments ayant le même nom et des contenus différents,
- Les attributs ne peuvent pas contenir de structures d'arbres, au contraire des éléments,
- Les attributs ne sont pas facilement extensibles, tandis qu'on peut toujours rajouter de nouveaux sous-éléments dans une hiérarchie sans changer les logiciels existants,
- Il n'est pas plus compliqué d'accéder à des sous-éléments qu'à des attributs car tous sont des nœuds dans l'arbre sous-jacent.

1.5.4. Attributs ou contenu ? (fin)

Avec les remarques précédentes, il est possible de construire ce genre de document :




```
<voiture id="649">
  <marque niveau="filiale">Dacia</marque>
  <marque niveau="generale">Renault</marque>
  <couleur>grise
    <nuance reference="EYP">gris banquise</nuance>
  </couleur>
</voiture>
```

Remarque : cela dépasse ce qu'on peut représenter dans une table de base de données. Ce serait le résultat d'une jointure

1.5.5. Associations

La représentation XML d'une association entre deux TE est un peu problématique quand les cardinalités ne sont pas (1,1) d'un côté. Les documents XML sont essentiellement hiérarchiques et représentent les informations uniquement vues d'une seule chose. Il faudrait donc mettre l'extension de la relation.

Lorsque l'une des cardinalités est (1,1), il est possible de construire une hiérarchie centrée sur l'un des TE. Voici par exemple, une relation de location entre (1,1) véhicule et θ à n clients : 

```
<voiture id="649">
  <marque>Renault</marque> ...
  <location date="15/11/2019"><nom>Dupond</nom> ...</location>
  <location date="25/01/2020"><nom>Dupont</nom> ...</location>
</voiture>
```

Semaine 2

Validation d'un document XML

La validation permet de vérifier la structure et le contenu d'un document XML avant de commencer à le traiter : les bons éléments avec les bons attributs présents au bon endroit.

Nous allons voir deux techniques :

- La plus simple basée sur les *Document Type Definitions* (DTD) venant de la norme SGML,
- La plus complète basée sur des *XML Schema*.

2.1. Validité d'un document

2.1.1. Introduction

Il y a deux niveaux de correction pour un document XML :

- Un document XML *bien formé* (*well formed*) respecte les règles syntaxiques d'écriture XML : écriture des balises, imbrication des éléments, entités, etc. C'est la plus facile des vérifications.
- Un document *valide* respecte des règles supplémentaires sur les noms, attributs et organisation des éléments.

La validation est cruciale pour une entreprise telle qu'une banque qui gère des transactions représentées en XML. S'il y a des erreurs dans les documents, cela peut compromettre l'entreprise. Il vaut mieux être capable de refuser un document invalide plutôt qu'essayer de le traiter et pâtir des erreurs qu'il contient.

2.1.2. Processus de validation

D'abord, il faut disposer d'un fichier contenant des règles. Il existe plusieurs langages pour faire cela : DTD, XML Schemas, RelaxNG et Schematron. Ces langages modélisent des règles de validité plus ou moins précises et d'une manière plus ou moins lisible.

Chaque document XML est comparé à ce fichier de règles, à l'aide d'un outil de validation : `xmlstarlet`, `xmllint`, `rnv`...

La validation indique que soit le document est valide, soit il contient des erreurs comme : tel attribut de tel élément contient une valeur interdite par telle contrainte, il manque tel sous-élément dans tel élément, etc.

2.2. Document Type Definitions (DTD)

2.2.1. Présentation

Un *Document Type Definitions* est une liste de règles définies au début d'un document XML pour permettre sa validation avant sa lecture. Elle est déclarée par un élément spécial DOCTYPE juste après le prologue et avant la racine :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE itineraire ... >
<itineraire nom="essai">
  <etape distance="0km">départ</etape>
  <etape distance="1km">tourner à droite</etape>
</itineraire>
```

Les DTD sont issues de la norme SGML et n'ont pas la syntaxe XML.

2.2.2. Intégration d'une DTD

Une DTD peut être :

- interne, intégrée au document. C'est signalé par un couple [] :

```
<!DOCTYPE itineraire [
    ...
]>
```

- externe, dans un autre fichier, signalé par SYSTEM suivi de l'URL du fichier :

```
<!DOCTYPE itineraire SYSTEM "itineraire.dtd">
```

- mixte, il y a à la fois un fichier et des définitions locales :

```
<!DOCTYPE itineraire SYSTEM "itineraire.dtd" [
    ...
]>
```

2.2.3. Outils de validation d'un document avec DTD

Deux commandes permettent de valider un document : [xmlstarlet](#) et [xmllint](#).

- Pour vérifier la syntaxe d'un document XML :
 - `xmlstarlet val --well-formed -e document.xml`
 - `xmllint --noout document.xml`
- Pour valider un document par rapport à une DTD interne :
 - `xmlstarlet val --embed -e document.xml`
 - `xmllint --valid --noout document.xml`
- Pour valider un document par rapport à une DTD externe :
 - `xmlstarlet val --dtd document.dtd -e document.xml`
 - `xmllint --dtdvalid document.dtd --noout document.xml`

2.2.4. Contenu d'une DTD

Une DTD contient des règles comme celles-ci :


```
<!ELEMENT itineraire (etape+)>
<!ATTLIST itineraire nom CDATA #IMPLIED>
<!ELEMENT etape (#PCDATA)>
<!ATTLIST etape distance CDATA #REQUIRED>
```

Ce sont des règles de construction :

- des éléments (ELEMENT) : leur nom et le contenu autorisé,
- des attributs (ATTLIST) : nom et options.

Voici maintenant les explications.

2.2.5. Racine du document

Le nom présent après le mot-clé DOCTYPE indique la racine du document. C'est un élément qui est défini dans la DTD. 

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE itineraire [
  <!ELEMENT itineraire (etape+)>
  <!ATTLIST itineraire nom CDATA #IMPLIED>
  <!ELEMENT etape (#PCDATA)>
  <!ATTLIST etape distance CDATA #REQUIRED>
]>
<itineraire nom="essai">
  <etape distance="0km">départ</etape>
  <etape distance="1km">tourner à droite</etape>
</itineraire>
```


2.2.6. Définition d'un élément

La règle `<!ELEMENT nom contenu>` permet de définir un élément : son nom et ce qu'il peut y avoir entre ses balises ouvrante et fermante. La définition du *contenu* peut prendre différentes formes :

- EMPTY : signifie que l'élément doit être vide,
- ANY : signifie que l'élément peut contenir n'importe quels éléments et textes (rien ne sera testé),
- (#PCDATA) : signifie que l'élément ne contient que des textes,
- (*définitions de sous-éléments*) : spécifie les sous-éléments qui peuvent être employés, voir plus loin.

Les DTD ne sont pas exigeantes sur les types des données.

2.2.7. Exemple de contenus

Voici un exemple d'éléments simples : 

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE itineraire [
  <!ELEMENT itineraire (boucle?, etape+, variante*)>
  <!ELEMENT boucle EMPTY>
  <!ELEMENT etape (#PCDATA)>
  <!ELEMENT variante ANY>
]>
<itineraire>
  <boucle/>
  <etape>départ</etape>
  <etape>tourner à droite</etape>
  <variante>
    <etape>départ</etape><etape>tourner à gauche</etape>
  </variante>
</itineraire>
```

2.2.8. Définition de sous-éléments

On arrive à la définition de sous-éléments. C'est une liste ordonnée dans laquelle chaque sous-élément peut être suivi d'un joker parmi * + ? identiques à ceux de `egrep` pour indiquer des répétitions.

```
<!ELEMENT itineraire (boucle?, etape+, variante*)>
```

L'élément `<boucle>` est en option, il doit être suivi d'un ou plusieurs `<etape>` puis éventuellement de plusieurs éléments `<variante>` ou aucun.

La liste peut contenir des alternatives exclusives notées (*contenu1* | *contenu2* | ...) comme avec `egrep` :

```
<!ELEMENT informations (topoguide | carte)>
```

signifie que l'élément `<information>` peut contenir soit un enfant `<topoguide>`, soit un enfant `<carte>`.

2.2.9. Contenus alternatifs

On peut grouper plusieurs séquences avec des parenthèses pour spécifier ce qu'on désire :

```
<!ELEMENT personne (titre?, (nom,prenom+) | (prenom+,nom))>
```

Pour finir sur les contenus, on peut aussi indiquer qu'ils peuvent contenir du texte ou des sous-éléments :

```
<!ELEMENT etape (#PCDATA | waypoint)* >
```

Cela permet de valider ces éléments :


```
<etape>avancer tout droit</etape>
<etape><waypoint lon="3.1" lat="48.2"/></etape>
<etape><waypoint lon="3.2" lat="48.1"/>aller au phare</etape>
```

2.2.10. Définition des attributs

On emploie la règle `<!ATTLIST elem attr type valeur ...>` : le nom de l'élément concerné, le nom de l'attribut, son type (voir plus loin) et sa valeur. Le mot clé `#REQUIRED` en tant que valeur indique que l'attribut est obligatoire, `#IMPLIED` qu'il est optionnel, et si on fournit une chaîne "...", c'est la valeur par défaut.

```
<!ELEMENT waypoint EMPTY>
<!ATTLIST waypoint
  lon      CDATA  #REQUIRED
  lat      CDATA  #REQUIRED
  ele      CDATA  #IMPLIED
  precision CDATA  "50m"
  source   (gps|utilisateur|carte) "carte">
```

NB: le type de l'attribut n'est pas représenté finement par les DTD, voir les schémas plus loin.

2.2.11. Types d'attributs

Il y a de plusieurs types possibles, voir [cette page](#) :

CDATA l'attribut peut prendre n'importe quelle valeur texte. Il n'y a malheureusement pas de vérification sur ce qui est fourni. Ne pas confondre avec `#PCDATA`.

(mot1|mot2|...) Cela force l'attribut à avoir l'une des valeurs de l'énumération.

ID l'attribut est un identifiant XML, sa valeur doit être une chaîne (pas un nombre) unique parmi tous les autres attributs de type ID du document.

IDREF l'attribut doit être égal, dans le document XML, à l'identifiant d'un autre élément.

NMTOKEN l'attribut doit être un nom d'élément ou d'attribut bien formé.

2.2.12. Définition d'entités (rappel et précisions)

Les entités sont des symboles qui représentent des morceaux d'arbre XML ou des textes.

```
...
<!ENTITY copyright "© IUT Lannion 2020">
<!ENTITY depart "<etape>Point de départ</etape>">
<!ENTITY equipement SYSTEM "equipement.xml">
]>
<itineraire>
  <auteur>&copyright;</auteur>
  &equipement;
  &depart;
</itineraire>
```

Pour valider et visualiser un document avec ses entités remplacées :

```
xmllint --valid --noent document.xml
```

2.2.13. Entités paramètres

Il est possible de définir une entité utilisée dans la DTD elle-même. La syntaxe est légèrement différente. Par exemple :

```
<!ENTITY % reference "(auteur, titre, date)">
<!ELEMENT livre (domaine, %reference;, ISBN, prix)>
<!ELEMENT email (%reference;, destinataires)>
```

Notez le % entre ENTITY et son nom, également la référence d'entité s'écrit %nom; et pas &nom;

De même pour les valeurs possibles d'un attribut. On a le choix de mettre les parenthèses dans l'entité ou dans son emploi :

```
<!ENTITY % taillepapier "A3|A4|A5">
<!ATTLIST lettre taille (%taillepapier;|USLETTER) #REQUIRED>
<!ATTLIST livre taille (%taillepapier;) #IMPLIED>
```

2.3. XML Schemas

2.3.1. Présentation

Les [Schémas XML](#) sont une norme W3C pour spécifier le contenu d'un document XML. Ils sont écrits en XML et permettent d'indiquer les conditions de validité beaucoup plus finement.

Voici un exemple de schéma pour se faire une idée :



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="reference" type="ElemReference" />
  <xsd:complexType name="ElemReference">
    <xsd:sequence>
      <xsd:element name="titre" type="xsd:string" />
      <xsd:element name="auteur" type="xsd:string" />
      <xsd:element name="ISBN" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

2.3.2. Association entre un document et un schéma local

Pour attribuer un schéma de validation local à un document XML, on peut ajouter un attribut situé dans un *namespace* spécifique :



```
<?xml version="1.0"?>
<reference
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="reference.xsd">
  <titre>Comprendre XSLT</titre>
  <auteur>Bernd Amann et Philippe Rigaux</auteur>
  <ISBN>2-84177-148-2</ISBN>
</reference>
```

On valide le document par :

- `xmllint --schema schema.xsd --noout document.xml`
- `xmllstarlet val --xsd schema.xsd -e document.xml`

2.3.3. Association entre un document et un schéma public

Lorsque le schéma est public, mis sur un serveur, c'est un peu différent car il faut définir un *namespace* et l'URL d'accès : 

```
<?xml version="1.0"?>
<reference
  xmlns="http://www.iut-lannion.fr"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.iut-lannion.fr reference.xsd">
  <titre>Comprendre XSLT</titre>
  <auteur>Bernd Amann et Philippe Rigaux</auteur>
  <ISBN>2-84177-148-2</ISBN>
</reference>
```

Et il faut ajouter les attributs `xmlns` et `targetNamespace` valant le même *namespace* à la racine du schéma, voir [cet exemple](#).

2.3.4. Principes généraux des Schémas XML

Comme une DTD, un schéma permet de définir des éléments, leurs attributs et leurs contenus. Mais il y a une notion de typage beaucoup plus forte qu'avec une DTD. Avec un schéma, il faut définir les types de données très précisément :

- la nature des données : chaîne, nombre, date, etc.
- les contraintes qui portent dessus : domaine de définition, expression régulière, etc.

Avec ces types, on définit les éléments :

- noms et types des attributs
- sous-éléments possibles avec leurs répétitions, les alternatives, etc.

C'est tout cela qui complique beaucoup la lecture d'un schéma.

2.3.5. Structure générale d'un schéma

Un schéma est contenu dans un arbre XML de racine `<xsd:schema>`. Le contenu du schéma définit les éléments qu'on peut trouver dans le document. Voici un squelette de schéma :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="itineraire" type="ElemItineraire" />
  ... définition du type ElemItineraire ...
</xsd:schema>
```

Il valide le document partiel suivant :

```
<?xml version="1.0"?>
<itineraire>
  ...
</itineraire>
```

2.3.6. Remarque importante

On peut aussi écrire le schéma précédent ainsi :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="itineraire">
    ... définition de l'élément ...
  </xsd:element>
</xsd:schema>
```

Cette écriture ne dissocie pas les définitions de l'élément et du type. On met la définition en tant que contenu de l'élément.

C'est acceptable pour de tout petits documents, mais ça peut vite devenir illisible.

2.3.7. Définition d'éléments

Un élément `<nom>contenu</nom>` du document est défini par un élément `<xsd:element name="nom" type="TypeContenu">` dans le schéma.

Dans l'exemple suivant, le type est `xsd:string`, c'est du texte quelconque (équivalent à `#PCDATA` dans une DTD) :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="message" type="xsd:string"/>
</xsd:schema>
```

Ce schéma valide le document suivant :

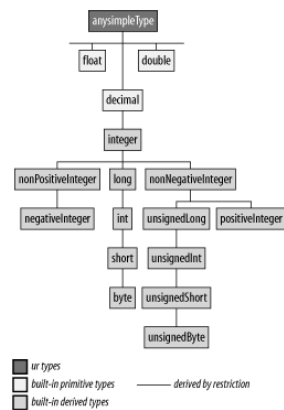


Figure 6: Types numériques

```
<?xml version="1.0"?>
<message>Tout va bien !</message>
```

2.3.8. Définition de types de données

L'exemple précédent indique que l'élément `<message>` doit avoir un contenu de type `xsd:string`, c'est à dire du texte. Ce type est un « type simple ». Il y a de nombreux [types simples prédéfinis](#), dont :

- chaîne :
 - `xsd:string` est le type le plus général
 - `xsd:token` vérifie que c'est une chaîne nettoyée des sauts de lignes et espaces d'indentation
- date et heure :
 - `xsd:date` correspond à une chaîne au format AAAA-MM-JJ
 - `xsd:time` correspond à HH:MM:SS.s
 - `xsd:datetime` valide AAAA-MM-JJTHH:MM:SS, on doit mettre un T entre la date et l'heure.

2.3.9. Types de données (suite)

- nombres :
 - `xsd:float`, `xsd:decimal` valident des nombres réels
 - `xsd:integer` valide des entiers
 - il y a de nombreuses variantes comme `xsd:nonNegativeInteger`, `xsd:positiveInteger`...

2.3.10. Types de données (suite)

- autres :
 - `xsd:ID` pour une chaîne identifiante, `xsd:IDREF` pour une référence à une telle chaîne
 - `xsd:boolean` permet de n'accepter que `true`, `false`, `1` et `0` comme valeurs dans le document.
 - `xsd:base64Binary` et `xsd:hexBinary` pour des données binaires.
 - `xsd:anyURI` pour valider des URI (URL ou URN).

2.3.11. Restrictions sur les types

Lorsque les types ne sont pas suffisamment contraints et risquent de laisser passer des données fausses, on peut rajouter des contraintes. Elles sont appelées *facettes* (*facets*).

Dans ce cas, on doit définir un type `simpleType` et lui ajouter des restrictions. Voici un exemple :

```
<xsd:element name="temperature" type="TypeTemperature" />

<xsd:simpleType name="TypeTemperature">
  <xsd:restriction base="xsd:decimal">
    <xsd:minInclusive value="-30"/>
    <xsd:maxInclusive value="+40.0"/>
  </xsd:restriction>
</xsd:simpleType>
```

2.3.12. Définition de restrictions

La structure d'une restriction est :


```
<xsd:restriction base="type de base">
  <xsd:CONTRAINTE value="PARAMETRE"/>
  ...
</xsd:restriction>
```

```
<xsd:simpleType name="TypeNumeroSecu">
  <xsd:restriction base="xsd:string">
    <xsd:whiteSpace value="collapse"/>
    <xsd:pattern value="[12][0-9]{12}([0-9]{2})?" />
  </xsd:restriction>
</xsd:simpleType>
```

Les contraintes qu'on peut mettre dépendent du type de données. Il y a une hiérarchie entre les types qui fait que par exemple le type nombre hérite des restrictions possibles sur les chaînes.


2.3.13. Restriction communes à tous les types

Ces restrictions (*facettes*) sont communes à tous les types :

- longueur de la donnée : `xsd:length`, `xsd:maxLength`, `xsd:minLength`. Ces contraintes vérifient que le texte présent dans le document a la bonne longueur.
- expression régulière étendue (egrep) : `xsd:pattern`. C'est une contrainte très utile pour vérifier toutes sortes de données : 

```
<xsd:simpleType name="TypeTemperature">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[-+]?[1-9][0-9]?°C"/>
    <xsd:whiteSpace value="collapse"/>
  </xsd:restriction>
</xsd:simpleType>
```

2.3.14. Restrictions communes (suite)

- `xsd:whiteSpace` indique ce qu'on doit faire avec les caractères espaces, tabulation et retour à la ligne éventuellement présents dans les données à vérifier :
 - `value="preserve"` : on les garde tels quels
 - `value="replace"` : on les remplace par des espaces
 - `value="collapse"` : on les supprime tous (à utiliser avec un motif).
- énumération de valeurs possibles : 

```
<xsd:simpleType name="TypeFreinsVélo">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="disque"/>
    <xsd:enumeration value="patins"/>
    <xsd:enumeration value="rétropédalage"/>
  </xsd:restriction>
</xsd:simpleType>
```

2.3.15. Restrictions sur les dates et nombres

Les dates et nombres possèdent quelques contraintes sur la valeur exprimée :

- bornes inférieure et supérieure :
 - `xsd:minExclusive` et `xsd:minInclusive`
 - `xsd:maxExclusive` et `xsd:maxInclusive`
- En plus de ces facettes, les nombres permettent de vérifier le nombre de chiffres :
 - `xsd:totalDigits` : vérifie le nombre de chiffres total (partie entière et fractionnaire, sans compter le point décimal)
 - `xsd:fractionDigits` : vérifie le nombre de chiffres dans la partie fractionnaire.

2.3.16. Types à alternatives

Comment valider une donnée qui pourrait être de plusieurs types possibles, par exemple, valider les deux premiers éléments et refuser le troisième :

```
<couleur>Chartreuse</couleur>
<couleur>#7FFF00</couleur>
<couleur>02 96 46 93 00</couleur>
```

Si on déclare l'élément `<couleur>` comme acceptant n'importe quel contenu, on ne pourra rien vérifier :

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

2.3.17. Types à alternatives (suite)

Alors on crée un « type à alternatives » qui est équivalent à plusieurs possibilités, par exemple `xsd:token` ou `xsd:hexBinary`. Attention, ce n'est pas comme déclarer une énumération de *valeurs possibles*. Ici, on parle de *types possibles*.

Pour exprimer qu'un type peut correspondre à plusieurs autres types, il faut le définir en tant que `<xsd:union>` et mettre les différents types possibles dans l'attribut `memberTypes` :

```
<xsd:simpleType name="TYPE_ALTERNATIF">
  <xsd:union memberTypes="TYPE1 TYPE2 ..."/>
</xsd:simpleType>
```

Les types possibles sont séparés par un espace.

2.3.18. Exemple de type à alternatives

Voici un exemple pour les couleurs :



```
<xsd:simpleType name="TypeCouleurs">
  <xsd:union memberTypes="TypeCouleursNom TypeCouleursHex"/>
</xsd:simpleType>

<xsd:simpleType name="TypeCouleursNom">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z][a-z] +"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="TypeCouleursHex">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="#[0-9A-F]{6}"/>
  </xsd:restriction>
</xsd:simpleType>
```

2.3.19. Données de type liste

Comment contraindre un élément à contenir des données sous forme de liste (séparées par des espaces), par exemple



```
<departements>22 29 35 44 56</departements>
```

C'est facile à l'aide d'un « type liste » basé sur un type simple, ici `xsd:integer`. C'est une construction en deux temps :

- il faut le type de base. Dans l'exemple, c'est un type qui définit ce qu'est un numéro de département correct, une restriction d'entier positif à deux chiffres.
- on l'emploie dans une définition de type `<xsd:list>` :

```
<xsd:simpleType name="TYPE_LISTE">
  <xsd:list itemType="TYPE_BASE"/>
</xsd:simpleType>
```


2.3.20. Exemple de liste

Voici la feuille XSD complète :



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

  <xsd:element name="departements" type="ElemListeDepartements"/>

  <xsd:simpleType name="ElemListeDepartements">
    <xsd:list itemType="TypeDepartement"/>
  </xsd:simpleType>

  <xsd:simpleType name="TypeDepartement">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:totalDigits value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

2.3.21. Contenu d'éléments

On revient maintenant sur les éléments. Nous avons vu comment définir un élément contenant un texte, un nombre, etc. :

```
<xsd:element name="NOM" type="TYPE"/>
```

Ça définit une balise <NOM> pouvant contenir des données du type indiqué par TYPE :

```
<?xml version="1.0"?>
<NOM>données correspondant à TYPE</NOM>
```

Comment définir un élément dont le contenu peut être d'autres éléments, ainsi que des attributs ? En fait, c'est la même chose, sauf que le type est « complexe ». Un type complexe peut contenir des sous-éléments et des attributs.

2.3.22. Type complexe

Pour modéliser un élément <personne> ayant deux éléments enfants <prénom> et <nom>, il suffit d'écrire ceci :



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="personne" type="ElemPersonne"/>
  <xsd:complexType name="ElemPersonne">
    <xsd:all>
      <xsd:element name="prénom" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

```
        <xsd:element name="nom" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
</xsd:schema>
```

La structure `<xsd:all>` contient une liste d'éléments qui doivent se trouver dans le document à valider. Il y a d'autres structures.

2.3.23. Contenu d'un type complexe


Un `<xsd:complexType>` peut contenir trois sortes d'enfants :

```
<xsd:complexType name="ElemPersonne">
    <xsd:sequence> ou <xsd:choice> ou <xsd:all>...
</xsd:complexType>
```

Les enfants peuvent être :

- `<xsd:sequence>`éléments...`</xsd:sequence>` : ces éléments doivent arriver dans cet ordre
- `<xsd:choice>`éléments...`</xsd:choice>` : le document à valider doit contenir l'un des éléments
- `<xsd:all>`éléments...`</xsd:all>` : le document à valider doit contenir certains de ces éléments et dans l'ordre qu'on veut.

2.3.24. Exemple de séquence

Dans cet exemple, les éléments `<personne>`, `<numero>`, `<rue>`, `<cpostal>` et `<ville>` doivent se suivre dans cet ordre : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="adresse" type="ElemAdresse"/>
    <xsd:complexType name="ElemAdresse">
        <xsd:sequence>
            <xsd:element name="personne" type="ElemPersonne"/>
            <xsd:element name="numero" type="xsd:integer"/>
            <xsd:element name="rue" type="xsd:string"/>
            <xsd:element name="cpostal" type="xsd:integer"/>
            <xsd:element name="ville" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

2.3.25. Exemple de choix

Pour représenter une limite temporelle, par exemple la date de fin d'une garantie, soit on mettra un élément `<date_fin>` soit un élément `<durée>` : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="limite" type="ElemLimiteTemps"/>
  <xsd:complexType name="ElemLimiteTemps">
    <xsd:choice>
      <xsd:element name="date_fin" type="xsd:date"/>
      <xsd:element name="durée" type="xsd:positiveInteger"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>
```

2.3.26. Imbrication de structures

On peut imbriquer plusieurs structures pour définir des éléments à suivre et en option :

```
<xsd:complexType name="ElemPersonne">
  <xsd:sequence>
    <xsd:element name="prénom" type="xsd:string"/>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:choice>
      <xsd:element name="age" type="xsd:string"/>
      <xsd:element name="date_naiss" type="xsd:date"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Par contre, on ne peut pas faire de mélange avec `<xsd:all>`.

2.3.27. Nombre de répétitions

Dans le cas de la structure `<xsd:sequence>`, il est possible de spécifier un nombre de répétition pour chaque sous-élément.

```
<xsd:complexType name="ElemPersonne">
  <xsd:sequence>
    <xsd:element name="prénom" type="xsd:string"
      minOccurs="1" maxOccurs="2"/>
    <xsd:element name="nom" type="xsd:string"
      minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

Par défaut, les nombres de répétitions min et max sont 1. Pour enlever une limite sur le nombre maximal, il faut écrire `maxOccurs="unbounded"`.

2.3.28. Définition d'attributs

Les attributs se déclarent dans un `<xsd:complexType>` :

```
<xsd:complexType name="ElemPersonne">
  ...
  <xsd:attribute name="NOM" type="TYPE" [OPTIONS] />
</xsd:complexType>
```

nom le nom de l'attribut

type le type de l'attribut, ex: `xsd:string` pour un attribut quelconque

options mettre `use="required"` si l'attribut est obligatoire, mettre `default="valeur"` s'il y a une valeur par défaut.

2.3.29. Cas spéciaux

Plusieurs situations sont assez particulières et peuvent sembler très compliquées :

- éléments vides sans ou avec attributs
- éléments textes sans ou avec attributs
- éléments avec enfants sans ou avec attributs
- éléments avec textes et enfants sans ou avec attributs

Voici comment elles sont modélisées en XML Schémas.

2.3.30. Élément vide sans attribut

C'est le cas le plus simple :



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

  <xsd:element name="test" type="ElemTest"/>

  <xsd:complexType name="ElemTest"/>

</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :



```
<?xml version="1.0" encoding="utf-8"?>
<test/>
```

2.3.31. Élément vide avec attribut

On rajoute un attribut obligatoire :



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

  <xsd:element name="test" type="ElemTest"/>

  <xsd:complexType name="ElemTest">
    <xsd:attribute name="att" type="xsd:string" use="required"/>
  </xsd:complexType>

</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :



```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"/>
```

2.3.32. Élément texte sans attribut

Il suffit de définir l'élément en tant que `simpleType` avec un `<xsd:restriction>` pour définir son contenu :



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

  <xsd:element name="test" type="ElemTest"/>

  <xsd:simpleType name="ElemTest">
    <xsd:restriction base="xsd:integer"/>
  </xsd:simpleType>

</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :



```
<?xml version="1.0" encoding="utf-8"?>
<test>123</test>
```

2.3.33. Élément texte avec attribut

On doit faire ainsi :



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
  <xsd:complexType name="ElemTest">
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>
```

```
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok">456</test>
```

2.3.34. Éléments enfants sans attribut

C'est comme précédemment, par exemple une séquence :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
  <xsd:complexType name="ElemTest">
    <xsd:sequence>
      <xsd:element name="test1"/>
      <xsd:element name="test2" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test><test1/><test2>texte</test2></test>
```

2.3.35. Éléments enfants avec attribut

Pour valider des attributs sur l'élément parent :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
  <xsd:complexType name="ElemTest">
    <xsd:sequence>
      <xsd:element name="test1"/>
      <xsd:element name="test2" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="att" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"><test1/><test2>texte</test2></test>
```

2.3.36. Éléments enfants avec texte mélangé

Rajouter l'attribut `mixed="true"` à `<xsd:complexType>` :



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
  <xsd:complexType name="ElemTest" mixed="true">
    <xsd:sequence>
      <xsd:element name="test1"/>
      <xsd:element name="test2" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :



```
<?xml version="1.0" encoding="utf-8"?>
<test>texte<test1/>texte<test2>texte2</test2>texte</test>
```

Semaine 3

RelaxNG et XPath

Le cours de cette semaine présente deux mécanismes :

- RelaxNG encore un autre mécanisme de validation d'un document,
- XPath pour extraire des informations d'un document XML.

3.1. RelaxNG

3.1.1. Présentation

Nous revenons vers la validation de documents XML. RelaxNG (*Regular Language for XML Next Generation*) permet d'écrire des feuilles de validation de manière à la fois complète et agréable.

RelaxNG offre une syntaxe simple comme une DTD et des types aussi précis qu'un schéma.

En termes de performance, RelaxNG est bien supérieur aux Schémas XML. RelaxNG est adapté à la validation en masse de gros documents.

3.1.2. Exemple de document à valider

Voici un document modélisant des messages simples :



```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
  <message numero="1" date="2020-01-01">
    <dest>promo2020</dest>
    <dest bcc="oui">Pierre Nerzic</dest>
    <contenu>Bonne année !</contenu>
  </message>
  <message numero="2">
    <dest>promo2020</dest>
    <contenu>Bonne rentrée !</contenu>
  </message>
  ...
</messages>
```

3.1.3. DTD du document

Sa DTD est :




```
<!--ELEMENT messages ( message+ ) >

<!--ELEMENT message ( dest+, contenu ) >
<!--ATTLIST message numero CDATA #REQUIRED >
<!--ATTLIST message date CDATA #IMPLIED >

<!--ELEMENT dest ( #PCDATA ) >
<!--ATTLIST dest bcc (oui|non) "non" >

<!--ELEMENT contenu ( #PCDATA ) >
```

C'est très lisible mais les types sont trop simplistes :
CDATA = n'importe quoi.

3.1.4. XML Schema du document

Une partie de son schéma est :



```
<xsd:element name="messages" type="ElemMessages"/>

<xsd:complexType name="ElemMessages">
  <xsd:sequence>
    <xsd:element name="message" type="ElemMessage" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ElemMessage">
  <xsd:sequence>
    <xsd:element name="dest" type="TypeDest" maxOccurs="unbounded" />
    <xsd:element name="contenu" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="numero" type="xsd:positiveInteger" use="required" />
  <xsd:attribute name="date" type="xsd:date" />
  ...
```

3.1.5. Feuille RelaxNG

Sa feuille de validation RelaxNG s'écrit en « syntaxe compacte » :




```
element messages {
  element message {
    attribute numero { xsd:positiveInteger },
    attribute date { xsd:date }?,
    element dest {
      attribute bcc { "oui"|"non" }?,
      text
    },
  }+,
```

```
    element contenu { text }
  }*
}
```

Pour valider : `jing -c messages.rnc messages.xml`

La simplicité de la perfection.

3.1.6. Deux syntaxes pour RelaxNG

La syntaxe précédente s'appelle la *syntaxe compacte*, mais elle n'est pas au format XML. Alors il existe aussi une écriture XML : 

```
<?xml version="1.0" encoding="utf-8"?>
<rng:element name="messages"
  xmlns:rng="http://relaxng.org/ns/structure/1.0">
  <rng:oneOrMore>
    <rng:element name="message">
      <rng:attribute name="numero"><rng:data type="xsd:positiveInteger"/></rng:attribute>
      <rng:optional>
        <rng:attribute name="date"><rng:data type="xsd:date"/></rng:attribute>
      </rng:optional>
      ...
      <rng:element name="contenu"><rng:text /></rng:element>
    </rng:element>
  </rng:oneOrMore>
</rng:element>
```

3.1.7. Principes de RelaxNG, syntaxe compacte

Les éléments sont définis par :

element NOM { CONTENU } REPETITIONS

- Le *contenu* peut être :
 - des attributs définis par : `attribute NOM { TYPE }`
 - des éléments enfants, voir le transparent suivant
 - du texte : mot-clé `text` (mais voir page 51)
 - rien : mot-clé `empty`
- Les *répétitions* sont spécifiées par un joker style egrep : `* ? +`

```
element message {
  attribute numero { text },
  attribute date { text } ?,
  element dest { text } +,
  element contenu { text }
}*

```

3.1.8. Ordonnancement des éléments enfants

- Il faut séparer les enfants successifs par des virgules
- Pour accepter les sous-éléments d'un élément dans un ordre quelconque, il faut les séparer par `&` :

CONTENU1 & CONTENU2 ...

Exemple :

```
element personne {  
  element nom { text } & element prenom { text }  
}
```

- Si, au contraire il y a des alternatives exclusives, il faut écrire :

CONTENU1 | CONTENU2 ...

NB: en cas d'ambiguïté, il faut tout entourer de (...)

3.1.9. Exemple de successions dans le contenu

```
element annuaire {  
  element contact {  
    ( element personne {  
      element nom { text } & element prenom { text }  
    }  
    |  
    element entreprise { text }  
  ),  
  ( element email { text }+ | element telephone { text } )  
}*  
}
```

Voir page 53 une autre syntaxe plus claire.

3.1.10. Types XSD et contraintes

Le type des données quelconques est `text`. On peut aussi employer un type XML Schemas plus précis. Il suffit seulement de les préfixer par `xsd:` : `xsd:integer`, `xsd:date`, `xsd:string`, etc.

Il est possible de rajouter des contraintes, les mêmes que les restrictions des schémas. On les écrit à la suite entre {...} :

```
element message {  
  attribute numero { xsd:nonNegativeInteger },  
  attribute date {  
    xsd:date { minInclusive="2019-11-01" }  
  } ?,  
  element dest {
```

```
    xsd:string { pattern="[a-zA-Z_.]+@[a-zA-Z.]+"}  
  } +,  
  element contenu { xsd:string }  
}*
```

3.1.11. Alternatives

Lorsque RelaxNG vérifie un document XML, il normalise les valeurs (suppression des espaces avant et après) avant de regarder si elles sont du bon type. Il n'y a pas de normalisation quand ce sont des alternatives :

```
attribute type { "portable" | "fixe" | "fax" }
```

Cette règle autorise seulement la première de ces deux lignes :

```
<telephone type="portable">...  
<telephone type="    fixe    ">...
```

3.1.12. Types liste

Comme avec les schémas, on peut valider un élément contenant une séquence de valeurs séparées par des espaces :

```
element dimensions {  
  list { xsd:float, xsd:float, xsd:float, ("mm"|"cm") }  
}  
element vecteur { list { xsd:float+ } }
```

Ces règles autorisent des choses telles que :

```
<dimensions>3.35 6.87 -1.57 mm</dimensions>  
<vecteur>17.65 -98.2 374.2</vecteur>
```

3.1.13. Syntaxe compacte nommée

Au lieu d'imbriquer les définitions :

```
element NOM1 {  
  element NOM2 { TYPE2 },  
  element NOM3 { TYPE3 }  
}
```

On peut écrire :

```
start = NOM1
NOM1 = element * { NOM2, NOM3 }
NOM2 = element * { TYPE2 }
NOM3 = element * { TYPE3 }
```

Ici l'étoile désigne l'élément courant. Ne pas confondre avec le joker de répétition.

3.1.14. Exemple

La feuille précédente se ré-écrit en (mais PB: jing plante) :



```
start = element annuaire {
    contact*
}
contact = element * {
    (personne | entreprise), (email+ | telephone)
}
personne = element * {
    nom & prenom          # bug avec jing
}
entreprise = element * { text }
email = element * { text }
telephone = element * { text }
nom = element * { text }
prenom = element * { text }
```

3.2. XPath

3.2.1. Présentation

XPath est un mécanisme (syntaxe + fonctions) permettant d'extraire des informations d'un document XML. Par exemple, dans le document [messages.xml](#),

```
<messages>
  <message numero="1" date="2020-01-01">
    <dest>promo2020</dest>
    <dest bcc="oui">Pierre Nerzic</dest>
    <contenu>Bonne année !</contenu>
  </message>
  ...
```

extraire le contenu du message n°4 s'écrit ainsi en XPath :

```
/messages/message[@numero=4]/contenu
```

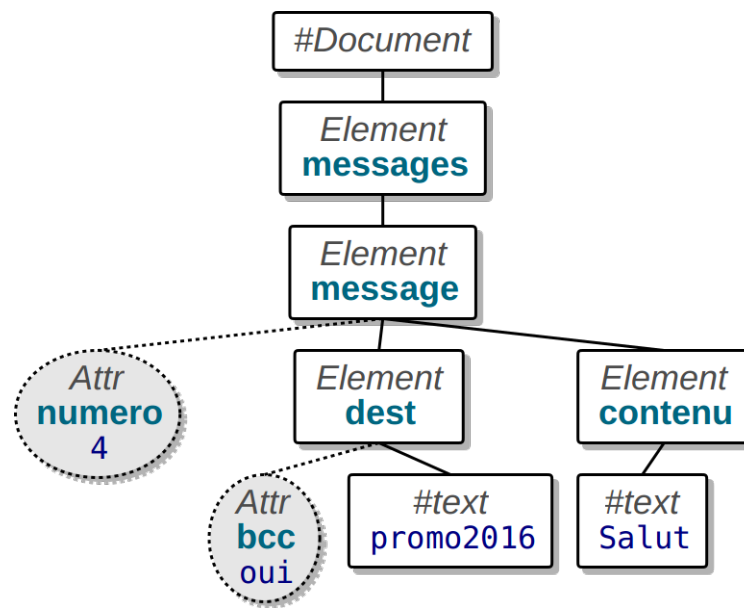


Figure 7: Arbre XML

3.2.2. Parcours d'arbre

XPath sert à extraire des informations dans un arbre XML.

[T] Soit ce document XML :



```
<?xml version="1.0"?>
<messages>
  <message numero="4">
    <dest bcc="oui">promo2020</dest>
    <contenu>Salut</contenu>
  </message>
</messages>
```

NB: le document complet contient tous les messages.

Voici l'arbre XML correspondant :

3.2.3. Principe général

Le but d'XPath est d'aller chercher les informations voulues dans le document XML. Ex: quels sont les destinataires du message n°2 ?

Cela se fait à l'aide d'un chemin d'accès qui ressemble beaucoup à un nom complet Unix, mais avec des conditions écrites entre [].

Exemple :

```
/messages/message[@numero="2"]/dest
```

C'est un peu comme un nom complet absolu dans Unix. Cependant, il y a énormément plus de possibilités pour écrire ces chemins et d'autre part, les chemins peuvent retourner plusieurs résultats.

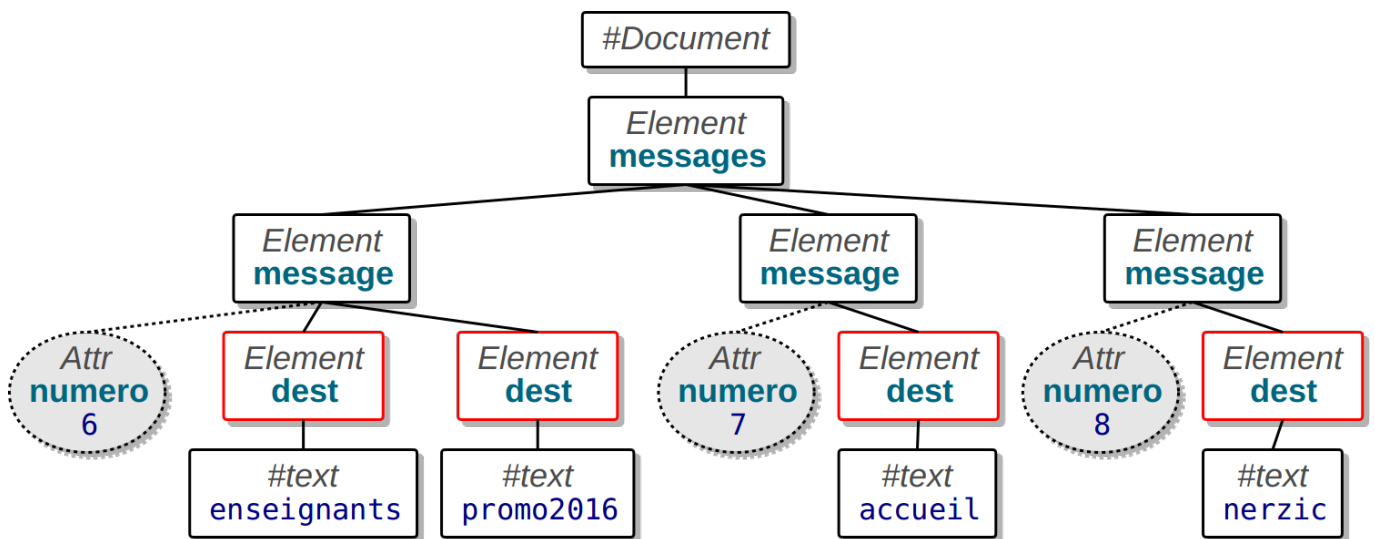


Figure 8: Arbre XML

3.2.4. Réponses multiples

Un point très important est qu'une expression XPath peut retourner plusieurs réponses. En effet, contrairement à Unix, un élément parent peut contenir plusieurs exemplaires du même élément enfant. Le chemin `/messages/message/dest` sélectionne 4 nœuds (rouge).

3.2.5. Évaluation d'une expression XPath

Pour évaluer une expression en ligne de commande, il y a :

- `xmlstarlet sel --template --value-of expression document.xml`
- `xmllint --xpath expression document.xml`
- `xmllint --shell document.xml` lance un mode interactif très intéressant :
 - `cat expression` : affiche les résultats de l'expression XPath
 - `grep mot` : cherche les occurrences du mot dans le document

La touche F9 dans *XML Copy Editor* permet de saisir une expression XPath.

Pour les navigateurs, je vous fournis [ce formulaire](#). Allez voir son source pour savoir comment évaluer du XPath en JavaScript.

3.2.6. XPath en JavaScript



```
var fichier = "messages.xml";
var xpath = "/messages/message/contenu/text()";
// demande du fichier XML au serveur
let requete = new XMLHttpRequest();
requete.onreadystatechange = function() {
  if (requete.readyState == 4 && requete.status == 200) {
    // évaluer l'expression XPath sur le XML reçu en réponse
    let nodes = xml.evaluate(xpath, requete.responseXML, null,
```

```
                XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);  
    // afficher les éléments XML dans le document HTML  
    while (node = nodes.iterateNext()) {  
        document.write(node.nodeValue + "<br/>");  
    }  
}  
}  
requete.open("GET", fichier); requete.send();
```

3.2.7. Structure d'une expression XPath simple

Une expression XPath est une suite d'étapes séparées par des séparateurs : [sep] étape1 sep étape2 sep étape3... Les étapes sont les noms des éléments dans lesquels il faut aller successivement.

Cependant cela dépend du séparateur employé :

- / Ce séparateur se comporte comme dans Unix : s'il est mis au début du chemin, il représente le document entier ; s'il est mis entre les étapes, c'est un simple séparateur.
- // Ce séparateur signifie de sauter un nombre quelconque d'éléments quelconques. C'est un peu comme si on écrivait `/*//*...` / un nombre indéfini de fois, y compris 0. S'il est au début du chemin, cela signifie alors que la première étape est à chercher n'importe où dans l'arbre.

3.2.8. Exemples

Voici quelques exemples de chemins :

- `/messages/message/dest` sélectionne tous les éléments `<dest>` de tous les éléments `<message>` de la racine `<messages>`.
- `//dest` sélectionne tous les éléments `<dest>` où qu'ils soient dans l'arbre. Ça donne le même résultat que l'exemple précédent parce qu'il n'y a pas de `<dest>` ailleurs.
- `/messages//contenu` sélectionne tous les éléments `<contenu>` situés sous la racine `<messages>`.

3.2.9. Attributs des éléments

Pour désigner un attribut et non pas un sous-élément, on met un `@` devant le nom de l'attribut.

Exemples :

- `/messages/message/@numero` sélectionne tous les nœuds Attributs nommés `numero` des éléments `<message>`.
- `//@numero` sélectionne tous les nœuds attributs portant ce nom n'importe où dans l'arbre.

NB: XPath retourne les nœuds attribut sélectionnés sous la forme `nom="valeur"`. Pour avoir seulement la valeur de l'un d'entre eux, il faut écrire `string(chemin)`. Par exemple, `string(//message[dest="promo2020"]/@numero)`. Attention `string()` exige qu'il n'y ait qu'un seul attribut sélectionné par l'expression XPath.

3.2.10. Autres étapes d'un chemin

D'autres étapes peuvent être employées :

- `.` désigne le nœud courant,

- `..` désigne le nœud parent,
- `*` désigne tous les éléments de ce niveau. C'est plus restreint que `//`.
- `|` regroupe les résultats de deux expressions XPath

Exemples :

- `/messages/*/@numero` sélectionne tous les nœuds Attributs nommés `numero` des éléments situés sous `<messages>`
- `//dest/@*` sélectionne tous les nœuds Attributs des éléments `<dest>` du document.
- `//message/dest|//message/contenu` sélectionne les éléments `<dest>` et `<contenu>` avec deux chemins complets.

3.2.11. Remarque sur l'opérateur d'alternative

Dans XPath 1.0, l'opérateur `|` permet seulement de réunir les résultats de deux expressions complètes et indépendantes. Contrairement au bon sens, l'expression suivante ne retourne pas les textes des `<dest>` et `<contenu>` trouvés dans les messages :

```
//message/dest|contenu/text()
```

Elle retourne seulement les éléments `<dest>` car la deuxième expression `contenu/text()` ne retourne rien.

- C'est seulement dans XPath 2.0 qu'on peut écrire :

```
//message/(dest|contenu)/text()
```

- En XPath 1.0, il faut malheureusement écrire :

```
//message/*[self::dest or self::contenu]/text()
```

3.2.12. Conditions sur les étapes

L'une des forces de XPath est de pouvoir rajouter des conditions appelées *prédicats* sur les étapes d'un chemin (éléments et attributs). Un prédicat se met entre `[...]` juste après l'élément dont il filtre l'un des enfants. On peut mettre plusieurs prédicats.

Exemple :

- `/messages/message[@numero=5]/contenu` sélectionne les `<contenu>` des messages dont l'attribut `numero` est 5.
- `//message[dest="promo2020"]/contenu` sélectionne la `<contenu>` des `<message>` ayant un sous-élément `<dest>` contenant la chaîne « promo2020 ».
- `//message[dest="iut"][@date="2020-01-01"]` sélectionne les `<message>` ayant un sous-élément `<dest>` contenant « iut » et un attribut `date` valant « 2020-01-01 ».

3.2.13. Syntaxe des conditions

Les conditions s'écrivent classiquement. On peut combiner des opérateurs logiques et d'autres conditions.

- Les comparaisons se font avec des expressions XPath qui portent sur le contenu du nœud courant,

- la notation `[index]` sélectionne l'élément ayant cet index (1 à n) dans la liste de son parent,
- le prédicat `[enfant]` est vrai si l'élément contient cet enfant.

Exemples :

- `//message[7]/contenu` sélectionne le `<contenu>` du 7e élément `<message>` du document.
- `//message[contenu and not(@date)]` sélectionne les `<message>` qui ont un `<contenu>` mais pas d'attribut `date`.

3.2.14. Opérateurs de comparaison

Pour écrire les prédicats, XPath propose ces opérateurs un peu différents de ceux du C :

- arithmétique : `+` `-` `*` `div` `mod` (et non pas `/` et `%`)
- comparaisons : `<` `<=` `=` `!=` `>=` `>` (et non pas `==`)
- logique : `and` `or` `not(condition)` (et non pas `&&`, `||` et `!`)

Exemples :

- `//message[not(@numero < 5 or @numero >= 9)]` sélectionne les `<message>` dont l'attribut `numero` est entre 5 et 8.
- `//message[@numero mod 5 = 0]` sélectionne les `<message>` dont l'attribut `numero` est un multiple de 5.

3.2.15. Fonctions XPath

XPath possède de très nombreuses [fonctions](#), dont :

- Fonctions sur les éléments :
 - `string(s)` retourne le texte de l'expression `s`
 - `position()` retourne l'index de l'élément dans son parent (premier = n°1)
 - `last()` retourne le n° du dernier élément dans son parent

Exemples :

- `string(/messages/message[2]/contenu)` retourne le contenu du 2e message.
- `/messages/message[position()<=3]` sélectionne les 3 premiers éléments `<message>` du document.
- `//dest[position()>last()-3]` sélectionne les `<dest>` qui sont parmi les trois derniers enfants de leur parent.

3.2.16. Fonctions XPath (suite)

Une fonction est particulièrement utile : `count(expression)`. Elle compte le nombre de nœuds XML (élément, attributs, textes...) sélectionnés par l'expression. On l'utilise dans des conditions.

Attention, `count()` compte les nœuds sélectionnés, chacun dans son parent *séparément*. Ça conduit à faire des erreurs si on croit que `count` peut regrouper différents comptages.

Exemples :

- `//message[count(dest)>2]` retourne les éléments `<message>` ayant plus de deux enfants `<dest>`.
- `//message[count(//dest)>2]` retourne tous les éléments `<message>` s'il y a plus de deux éléments `<dest>` quelque part dans le document.

3.2.17. Fonctions XPath (suite)

- Fonctions sur les chaînes :
 - `string-length(s)` retourne la longueur de la chaîne `s`
 - `concat(s1, s2,...)` concatène les chaînes passées
 - `substring(s,deb,lng)` retourne `lng` caractères de `s` à partir du n°`deb` (premier = 1)
 - `contains(s1, s2)` vrai si `s1` contient `s2`
 - `starts-with(s1,s2)` et `ends-with(s1,s2)`
 - `matches(s, motif)` vrai si `s` correspond au motif

Exemple :

- `//message[string-length(contenu)<=15 and not(starts-with(dest, "promo"))]/@numero`
retourne les numéros des messages dont le contenu ne fait pas plus de 15 caractères et aucun destinataire ne commence par « promo ».

3.2.18. Fonctions XPath (suite)

- Fonctions mathématiques :
 - `abs(nb)`, `ceiling(nb)`, `floor(nb)`, `round(nb)`
- Fonctions sur les dates et heures :
 - `year-from-dateTime(dt)`, `month-from-dateTime(dt)`, `day-from-dateTime(dt)`,
`hours-from-dateTime(dt)`, `minutes-from-dateTime(dt)`, `seconds-from-dateTime(dt)`
 - `year-from-date(d)`, `month-from-date(d)`, `day-from-date(d)`
 - `hours-from-time(t)`, `minutes-from-time(t)`, `seconds-from-time(t)`

3.2.19. Retour sur les composants d'un chemin

Un chemin XPath est constitué de `[sep] étape1 sep étape2 sep étape3...` Chaque étape est soit le nom d'un élément, soit `@` et le nom d'un attribut ; chacune suivie éventuellement d'un prédicat entre crochets :

```
/racine/element1[filtre1]/.../@attribut[filtre3]
```

Les étapes sont appelées *sélecteurs*. On peut employer des sélecteurs spéciaux comme :

text() sélectionne tous les nœuds texte sous l'élément courant, y compris tous ses descendants.
node() sélectionne tous les nœuds enfants de l'élément.

Exemple :

- `/messages/message/contenu/text()`

3.2.20. Axes

XPath permet de rajouter encore une « décoration » sur chaque étape, la *direction* dans laquelle aller à partir de l'étape courante. Cette direction est appelée *axe*. Cela donne la syntaxe :

```
/racine/axe1::element1[filtre1]/axe2::element2[filtre2]/...
```

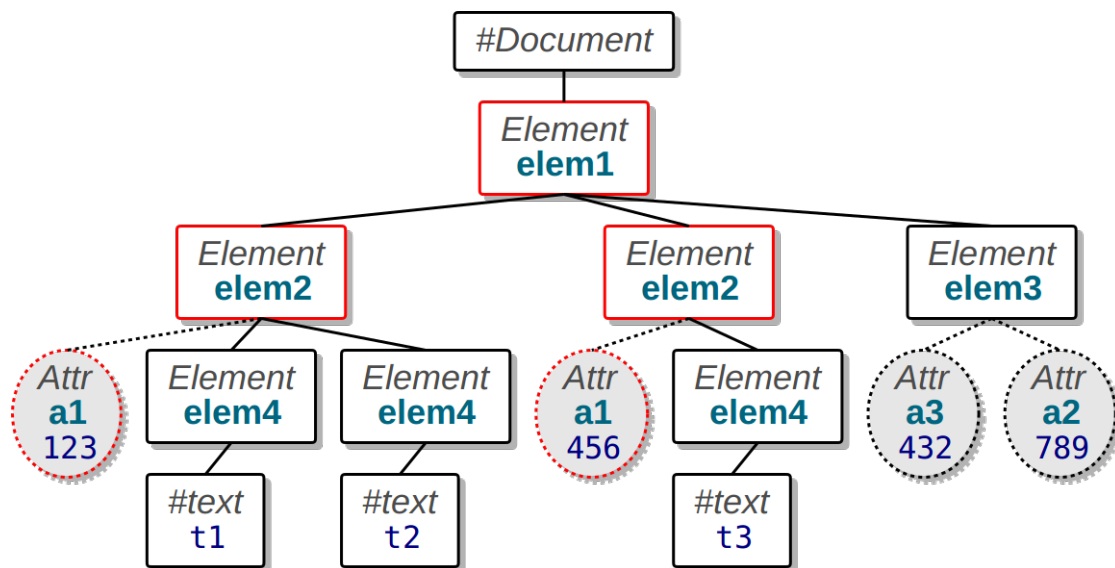


Figure 9: Arbre XML

Par défaut, on descend toujours vers les enfants du nœud courant au niveau de chaque étape. Cet axe s'appelle *child*.

Exemple, ces deux syntaxes signifient la même chose :

- `/messages/message/@numero`
- `/messages/child::message/attribute::numero`

D'autres axes existent. Pour les comprendre, il faut étudier l'algorithme de XPath.

3.2.21. Algorithme de XPath

`XPath(nœud parent, chemin)` est un algorithme récursif. Il sélectionne une liste de nœuds par un chemin partant d'un nœud parent qu'on appelle *contexte d'évaluation*. Au départ, le contexte c'est le document entier.

1. Si le chemin est vide, alors ajouter le nœud parent dans la réponse à la requête XPath globale
2. Sinon extraire la première étape du chemin : `axe::nom[prédicat]`
3. Passer en revue tous les nœuds (éléments ou attributs) du nœud parent définis par l'axe. Par exemple pour l'axe `child::` ce sont tous les nœuds enfants, pour l'axe `attribute::` ce sont les nœuds attributs
4. Si le nœud correspond à l'étape (nom et prédicat), alors faire un appel récursif à `XPath(nœud, reste du chemin)`

3.2.22. Exemple

Soit un document XML représentant un arbre de nœuds. Par exemple, celui-ci pour évaluer `"/elem1/elem2/@a1"` :

3.2.23. Exemple (suite)

On part du nœud parent égal au document entier, indiqué par le / initial : `XPath(#Document, "elem1/elem2/@a1")`.

1. On commence par prendre l'étape "elem1" puis on fait une boucle sur tous les enfants du document. Il n'y en a qu'un, c'est <elem1> qui correspond à cette étape. Donc on fait un appel récursif `XPath(<elem1>, "elem2/@a1")`.
2. On prend l'étape "elem2" puis on passe les enfants du nœud <elem1> en revue : il y a <elem2 a1="123">, <elem2 a1="456"> et <elem3 ...>. Les deux premiers correspondent à l'étape, donc, pour chacun d'eux, on fait un appel récursif :
 - a. `XPath(<elem2 a1="123">, "@a1")`
 - b. `XPath(<elem2 a1="456">, "@a1")`

3.2.24. Exemple (suite et fin)

Pour chacun des deux appels récursifs, il se passe la même chose :

3. On prend l'étape restante, "@a1". Comme elle désigne un attribut, c'est une boucle sur les attributs du nœud parent qui est faite. L'attribut a1 est présent et donc cela conduit à chaque fois à un nouvel appel récursif :
 - a. `XPath(#Attr<a1="123">, "")`
4. Le chemin est vide donc on rajoute le nœud attribut a1 dans la réponse finale.

Au final, la réponse contient les deux nœuds attribut a1 du document.

Ce qu'il faut comprendre, c'est l'importance des boucles de parcours des nœuds et l'appel récursif qui en résulte quand l'étape correspond au nœud.

3.2.25. Axes

L'axe définit quels sont les nœuds explorés à chaque étape. Voici quelques axes utiles à connaître parmi [ceux qui existent](#) :

child:: parcourir les nœuds enfants du contexte ; c'est l'axe utilisé par défaut.

descendant:: parcourir tous les nœuds enfants et petit-enfants ; ça revient un peu à utiliser `//`.

parent:: parcourir le nœud parent du contexte ; ça revient à utiliser `..` mais avec un test sur le parent voulu.

ancestor:: parcourir tous les nœuds parent et grand-parents.

preceding-sibling:: parcourir tous les nœuds frères précédents

following-sibling:: parcourir tous les nœuds frères suivants

attribute:: parcourir les nœuds attributs du contexte ; c'est l'axe par défaut pour une étape commençant par un @.

3.2.26. Exemples de chemins avec axes

- `/messages/message[last()]/child::contenu` retourne le contenu du dernier message du document.
- `/messages/message[@numero=7]/descendant::dest` sélectionne tous les nœuds situés sous le message n°7.
- `//message[@numero=5]/preceding-sibling::message` sélectionne les messages situés avant le n°5.
- `//dest[@bcc="oui"]/parent::node()` sélectionne le nœud parent d'un élément <dest> dont l'attribut bcc vaut oui.
- `//message[3]/attribute::numero` retourne l'attribut numéro du 3e message présent dans le document.

Semaine 4

Transformation d'un document

Le cours de cette semaine présente :

- L'utilisation de feuilles de style CSS,
- XSLT, un outil pour transformer un document XML.

Le but initial était de fournir une visualisation esthétique pour des documents XML, mais ça a évolué vers la transformation dans d'autres formats.

4.1. Feuilles de styles CSS

4.1.1. Feuille CSS pour un document XML

Normalement, un document XML ne peut pas être affiché dans un navigateur internet car ce n'est pas du HTML. On peut lui associer une feuille de style CSS pour spécifier l'affichage des éléments, ce qui permet de les voir correctement.

Il faut mettre un entête spécial dans le document XML qui spécifie la feuille de style à utiliser. Celle-ci définit l'apparence de chacun des éléments du document :

```
ELEMENT {  
    DECLARATIONS;  
    ...  
}
```

NB: cette solution n'est pas recommandée, car incapable d'afficher des documents un peu complexes.

4.1.2. Exemple de document XML

Voici le document `albums.xml` utilisé pour l'exemple :



```
<?xml version="1.0" encoding="utf-8"?>  
<?xml-stylesheet type="text/css" href="albums.css"?>  
<albums>  
  <album numero="1" serie="Tintin">  
    <titre>Tintin au pays des Soviets</titre>  
    <date>  
      <mois>septembre</mois><annee>1930</annee>  
    </date>
```

```
</album>
...
</albums>
```

4.1.3. Exemple de feuille de style CSS

Voici un extrait de la feuille de style `albums.css` (très simpliste) :



```
album {
    display: block;
}
album titre {
    display: inline-block;
    width: 500px;
    font-size: 16pt;
}
album mois, album annee {
    display: inline-block;
    width: 120px;
}
```

4.2. XSLT

4.2.1. Présentation

XSL est une norme pour définir des feuilles de styles en XML. XSLT est un langage permettant de transformer un document XML à l'aide d'expressions XPath écrites dans une feuille XSL.

Le premier but de XSLT est de permettre l'affichage d'un document XML dans un navigateur. Dans ce cas, la transformation consiste à générer du code HTML en fonction de ce qu'on trouve dans le document XML. XSLT est indispensable quand la structuration XML ne correspond pas à celle de HTML.

Par exemple, on souhaite afficher la liste des albums de Tintin dans un tableau HTML, en regroupant le mois et l'année dans une seule chaîne. Ces deux informations sont dans des éléments séparés. Ce n'est donc pas possible avec des styles CSS.

4.2.2. Exemple de feuille de style

Voici comment s'écrit la transformation en XSLT. On mélange des balises HTML avec des balises XSL :



```
<xsl:template match="/albums">
  <html><body>
    <h2>Albums de Tintin</h2>
    <table border="1">
      <xsl:for-each select="album[@serie='Tintin']">
        <tr>
```

```
<td><xsl:value-of select="titre"/></td>
<td><xsl:value-of select="date/annee"/></td>
</tr>
</xsl:for-each>
</table>
</body></html>
</xsl:template>
```

4.2.3. Entête d'une feuille XSLT

Une feuille XSLT doit commencer par ces lignes :

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

- La deuxième ligne identifie la norme de la feuille de style et définit le *namespace* `xsl:` de ses éléments.
- La troisième ligne indique le type de sortie : `xml`, `html` ou `text`. On peut rajouter les attributs `version="1.0"`, `encoding="UTF-8"` et `indent="yes"`.

Pour attribuer une feuille de style XSL à un document XML, il faut mettre ceci avant la racine du document :

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

4.2.4. Outil de transformation `xsltproc`

Pour appliquer une transformation à un fichier existant :

```
xsltproc feuille.xsl document.xml
```

Ça affiche le document transformé par la feuille. On peut rediriger la sortie vers un fichier.

Cette commande est capable d'utiliser les valeurs par défaut des attributs, telles que définies dans la DTD.

4.2.5. Principe général

XSLT prend un fichier XML en entrée, ainsi qu'une feuille XSL qui décrit les traitements à appliquer. En sortie, on récupère un document XML, HTML ou texte, selon la feuille XSL.

La feuille XSL contient des patrons (*templates*). Ce sont des sortes de couples (expression, contenu) : les parties du document XML qui correspondent à l'expression sont remplacées par le contenu.

- L'expression est écrite en XPath et porte sur l'arbre XML d'entrée. Elle indique quels sont les nœuds à remplacer par le contenu du patron.
- Le contenu du patron sont des éléments et textes qui sont mis à la place des nœuds sélectionnés.
- Les nœuds XML qui ne sont pas sélectionnés ne sont pas remplacés.

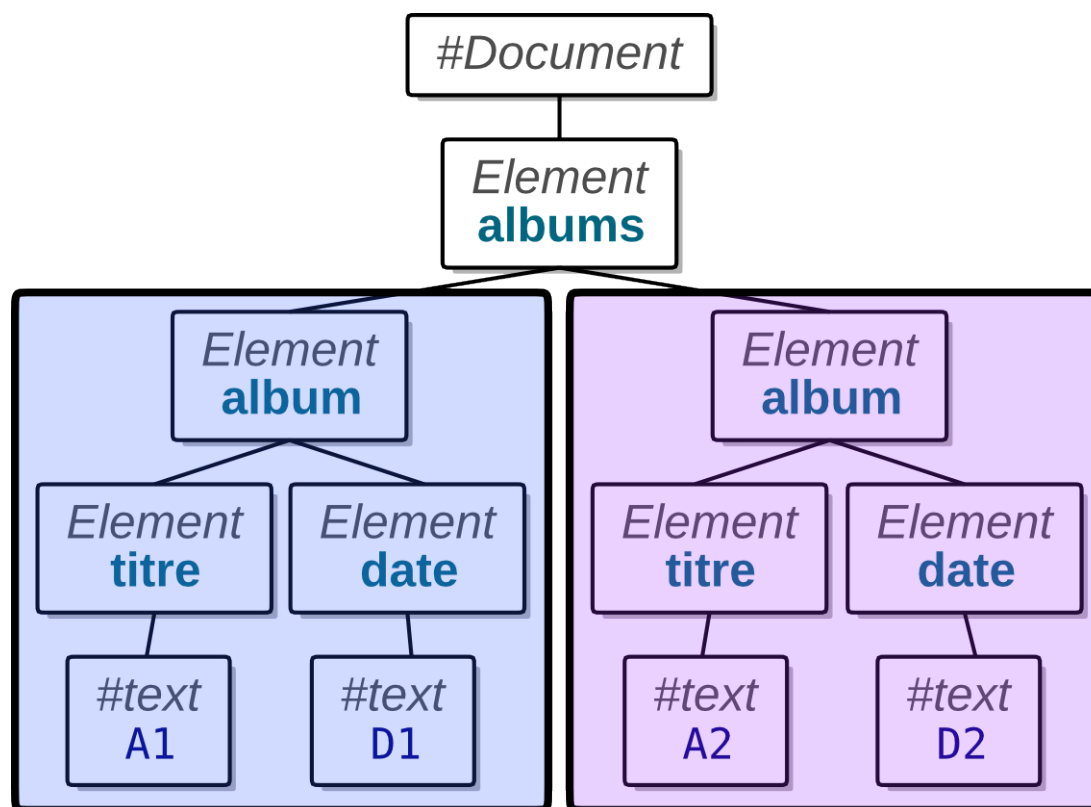


Figure 10: Arbre XML

4.2.6. Exemple de traitement

[T] Voici le document d'origine. On lui applique un patron dont l'expression est `/albums/album`.

Les éléments sélectionnés ont été remplacés.

Voir la figure 11, page 66.

4.2.7. Patrons

Voici la forme générale d'un patron :

```
<xsl:template match="XPATH">
  CONTENU
</xsl:template>
```

- XPATH est un chemin XPath, par exemple `/` pour désigner le document entier.
- CONTENU est un mélange d'éléments XSL et d'autres choses (éléments et textes) qui doivent respecter la syntaxe XML.

Dans le contenu, il peut y avoir des éléments XSL :

- `<xsl:value-of select="XPATH"/>` est remplacé par la valeur que renvoie l'expression XPath.
- `<xsl:text>texte...</xsl:text>` est remplacé par le texte tel quel.

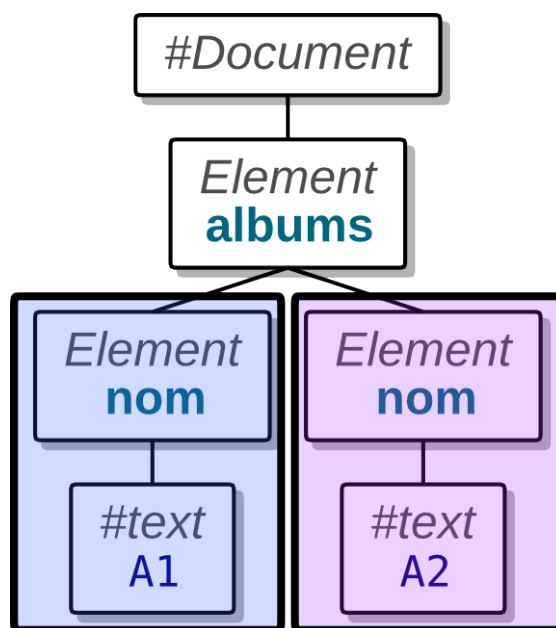


Figure 11: Arbre XML

4.2.8. Exemple de patron

Le patron de cette feuille remplace les éléments `<date>` du document `albums.xml` par `<sortie>mois
année</sortie>` :

```
<xsl:template match="/albums/album/date">
  <sortie>
    <xsl:value-of select="mois"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="annee"/>
  </sortie>
</xsl:template>
```

Notez l'espace dans l'élément `<xsl:text>`. Sans cette syntaxe, l'espace serait ignoré et absent de la sortie.

NB: ce patron ne peut pas être utilisé seul car il ne génère pas de racine en sortie : les autres éléments sortent aussi.

4.2.9. Créer des éléments et des attributs

Rajouter un élément est très facile. Il suffit de l'écrire tel quel, mais on peut aussi utiliser la balise `<xsl:element name="nom">`. Voici comment lui rajouter un attribut :

```
<xsl:template match="album">
  <xsl:element name="ouvrage">
    <xsl:attribute name="annee">
      <xsl:value-of select="date/annee"/>
    </xsl:attribute>
```

```
<xsl:value-of select="titre"/>
</xsl:element>
</xsl:template>
```

Ce exemple génère `<ouvrage annee="ANNEE">TITRE</ouvrage>` à partir des éléments `<album>` du document d'entrée.

4.2.10. value-of et copy-of

Ces directives permettent d'insérer une partie du contenu :

- `<xsl:value-of select="XPATH"/>` insère la valeur (texte) de l'expression XPATH
- `<xsl:copy-of select="XPATH"/>` insère les nœuds sélectionnés par l'expression XPATH

Exemple :



```
<xsl:template match="//album">
  <BD>
    <xsl:attribute name="nom">
      <xsl:value-of select="titre"/>
    </xsl:attribute>
    <xsl:copy-of select="date/annee"/>
  </BD>
</xsl:template>
```

4.2.11. Fonctions utiles

L'expression XPATH dans `<xsl:value-of select="XPATH"/>` peut employer des fonctions comme :

- `count(expression)` retourne le nombre de nœuds sélectionnés
- `sum(expression)` retourne la somme des nombres sélectionnés
- `string-length(chaîne)` : longueur de la chaîne
- `substring(chaîne,pos,lng)` : extrait lng caractères à partir de pos (≥ 1)
- `starts-with(chaîne,début)` vrai si la chaîne commence par début
- `end-with(chaîne,fin)` vrai si la chaîne finit par début
- `contains(chaîne,partie)` vrai si partie est dans la chaîne
- `concat(chaîne1,chaîne2...)` retourne les chaînes groupées

La liste complète est sur [cette page](#).

4.3. Structures de contrôle XSLT

4.3.1. Variables

Bien que ça ne soit pas trop la philosophie du langage XSLT, il est possible de stocker des données dans une variable puis l'utiliser plus tard. Les données sont n'importe quelle expression XPath et lors de l'utilisation, on peut s'appuyer dessus pour construire une expression plus complexe.

```
<xsl:variable name="NOM" select="VALEUR"/>
...
<xsl:value-of select="$NOM"/>
```

Exemple :

```
<xsl:variable name="MonAlbumPrefere"
  select="//album[date/annee=1949]"/>
...
<xsl:value-of select="$MonAlbumPrefere/titre"/>
```

4.3.2. Conditionnelles

Il est possible de générer du contenu conditionnel à l'aide de la structure XSL suivante : 


```
<xsl:if test="CONDITION">
  CONTENU
</xsl:if>
```

Les conditions sont écrites à la manière XPath, voir le cours 3.

La condition doit impérativement être correcte du point de vue XML, c'est à dire que les caractères < " > doivent être remplacés par leurs entités < ; " ; > et il faut mettre des espaces entre les valeurs et les opérateurs.

NB: il n'y a pas de *else*, mais on peut écrire deux <xsl:if> avec des conditions opposées, et voir <xsl:choose> plus loin.

4.3.3. Exemple de contenu conditionnel

Le patron suivant ne génère un élément <date> que si l'année est inférieure à 1940. Remarquez le signe < écrit sous forme d'entité et les espaces : 

```
<xsl:template match="/albums/album/date">
  <xsl:if test="annee &lt; 1940">
    <date>
      <xsl:value-of select="annee"/>
    </date>
  </xsl:if>
</xsl:template>
```

NB: ce patron ne gère pas la racine du document, donc tous les éléments autre que <date> sortent aussi.

4.3.4. Comment faire un *else* ?

Il existe une autre structure permettant toutes les combinaisons

```
<xsl:choose>
  <xsl:when test="condition1">
    ...contenu si condition1 est vraie
  </xsl:when>
  <xsl:when test="condition2">
    ...contenu si condition2 est vraie
  </xsl:when>
  ...
  <xsl:otherwise>
    ...contenu si aucune condition n'est vraie
  </xsl:otherwise>
</xsl:choose>
```

Seule la première condition vraie est appliquée.

4.3.5. Remarques sur les tests

Pour tester si un élément existe et/ou possède un contenu :

- `<xsl:if test="element">` est vrai si le nœud courant a un enfant appelé `<element>`, qu'il soit vide ou non
- `<xsl:if test="not(element)">` est vrai si le nœud courant n'a pas d'enfant appelé `<element>`
- `<xsl:if test="element != ''">` est vrai si le nœud courant a un enfant non-vide appelé `<element>`
- `<xsl:if test="element = ''">` est vrai si le nœud courant a un enfant vide appelé `<element/>`

4.3.6. Boucle sur les nœuds enfant

Lorsqu'on veut traiter tous les enfants d'un élément, on emploie l'élément `xsl:for-each` :

```
<xsl:for-each select="XPATH">
  CONTENU
</xsl:for-each>
```

Le document de sortie contiendra autant d'exemplaires du `CONTENU` que de nœuds sélectionnés par le chemin `XPATH`. Ce contenu est paramétré par le nœud courant de la boucle : `current()`, mais on fait généralement directement référence aux enfants du nœud désigné par le `select`.

4.3.7. Exemple de patron de type *boucle*

Le patron de cette feuille remanie le document `albums.xml` en `<liste><nom>titre</nom>...</liste>` :



```
<xsl:template match="/albums">
  <liste>
    <xsl:for-each select="album">
      <nom><xsl:value-of select="titre"/></nom>
    </xsl:for-each>
  </liste>
</template>
```

```
</xsl:for-each>
</liste>
</xsl:template>
```

Ce patron sélectionne l'élément `<albums>` et le remplace par `<liste>`, contenant des éléments `<nom>`, un par `<album>` du document d'entrée ; chaque élément `<nom>` contient le titre de l'album. Le `titre` à l'intérieur de la boucle est relatif à l'album courant.

4.3.8. Exemple de patron (suite)

Le même exemple peut être écrit autrement car il n'y a qu'un seul élément `<titre>` dans un album :



```
<xsl:template match="/">
  <liste>
    <xsl:for-each select="albums/album/titre">
      <nom><xsl:value-of select="current()" /></nom>
    </xsl:for-each>
  </liste>
</xsl:template>
```

Notez l'usage de la fonction `current()` pour désigner l'élément `<titre>`. Elle retourne le même résultat que `.` mais permet d'être employée dans une expression plus complexe.

4.3.9. Tri des itérations

La structure `xsl:for-each` itère sur une liste de nœuds du document. Cette liste est dans l'ordre du document, mais peut être triée selon un autre critère, voir [cette page](#) :



```
<xsl:template match="/">
  <xsl:for-each select="albums/album">
    <xsl:sort select="date/annee" data-type="number"/>
    <nom><xsl:value-of select="titre"/></nom>
  </xsl:for-each>
</xsl:template>
```

On peut spécifier :

- le type des données : `data-type="text|number|qname"`
- le sens du tri : `order="ascending|descending"`

4.3.10. Remarque sur les boucles

En XSL 2.0, il est possible d'itérer facilement sur des choses assez complexes. Par exemple :



```
<xsl:template match="/">
  <calendrier>
    <xsl:for-each select="distinct-values(//mois)">
      <mois>
        <xsl:value-of select="current()" />
        <xsl:text> : </xsl:text>
        <xsl:value-of
          select="count(//album[date/mois=current()])" />
        <xsl:text> albums</xsl:text>
      </mois>
    </xsl:for-each>
  </calendrier>
</xsl:template>
```

4.3.11. XSL 2.0

Hélas, XSL 2.0 utilise XPath 2.0 et n'est pas implémenté par xsltproc, ni de nombreux outils dont les navigateurs.

On doit donc trouver des rustines. Pour l'exemple précédent, on fait appel à une astuce proposée par [Steve Muench](#). Elle consiste à créer un index, c'est à dire des couples (valeur, liste des éléments ayant cette valeur). Par exemple, les couples (mois, liste des albums sortis ce mois).

La directive `<xsl:key name="nom_index" match="élément" use="clé">` demande à XLST de mémoriser un index appelé *nom_index* et contenant les éléments associés à la clé.

```
<xsl:key name="index" match="//mois" use="text()" />
```

Ici, la clé est le nom du mois et ça mémorise les éléments `<mois>`.

4.3.12. Index en XSL

L'instruction `<xsl:key ...>` précédente construit un tableau :

clé	éléments
janvier	<mois de l'album numéro="20">, <mois de l'album numéro="21">,...
mars	<mois de l'album numéro="31">
avril	<mois de l'album numéro="15">,...

Il faut bien comprendre que le même nom de mois est associé à plusieurs éléments distincts dans le document. Certains mois ne sont pas dans l'index car aucun album ne leur correspond.

La fonction `key('index', clé)` retourne la liste des éléments `<mois>` associés à la clé.

4.3.13. Indentifiants d'éléments

Une autre partie de l'astuce repose sur le fait que chaque élément possède un identifiant interne (caché) qui le distingue des autres. La fonction `generate-id()` permet d'obtenir cet identifiant pour l'élément courant.

La fonction `generate-id(elements)` permet d'obtenir l'identifiant des éléments fournis en paramètre.

L'astuce repose sur le fait qu'on va demander l'identifiant du premier élément indexé par le mois. On ne fera une boucle que pour cet élément, et pas pour les autres. Ainsi, il n'y aura qu'un seul exemplaire de chaque mois en sortie.

Hélas, on ne peut pas boucler sur les clés de l'index.

4.3.14. Groupement de Steve Muench

Ensuite, on fait une boucle sur les clés comme ceci :



```
<xsl:template match="/">
  <calendrier>
    <xsl:for-each
      select="//mois[
        generate-id()=generate-id(key('index',text())[1])] ">
      <mois>
        <xsl:value-of select="current()"/>
        <xsl:text> : </xsl:text>
        <xsl:value-of
          select="count(//album[date/mois=current()])"/>
        <xsl:text> albums</xsl:text>
      </mois>
    </xsl:for-each>
  </calendrier>
</xsl:template>
```

4.3.15. Traitement d'un document complexe

Pour traiter un document complet, il est fréquent de faire appel à plusieurs patrons : un pour le document entier / ou /racine et des patrons pour ses éléments. Dans ce cas, il faut explicitement signaler au patron racine d'appeler les patrons des éléments.

Cela se fait avec un élément `<xsl:apply-templates>` :

- sans attribut, il essaie tous les patrons sur tous les enfants,
- avec l'attribut `select="XPATH"`, il n'essaiera les autres patrons que sur les nœuds sélectionnés par le chemin XPATH.

Il faut signaler qu'il y a parfois des patrons par défaut dans les navigateurs, qui peuvent interférer avec votre traitement. Dans ce cas, il faut définir des patrons qui captent tous vos éléments.

4.3.16. Exemple de patrons imbriqués

Ces patrons extraient les albums de Tintin sous la forme `<tintin><nom>titre</nom>...</tintin>` :




```
<xsl:template match="/">
  <tintin>
    <xsl:apply-templates
      select="/albums/album[@serie='Tintin']"/>
    </tintin>
  </xsl:template>

  <xsl:template match="album">
    <nom><xsl:value-of select="titre"/></nom>
  </xsl:template>
```

C'est un peu plus simple qu'une boucle.

Semaine 5

XQuery et les bases de données XML

Le cours de cette semaine présente :

- XQuery qui est une extension de XPath,
- les bases de données XML et leur interrogation avec XQuery.

XQuery est un langage qui inclut XPath et qui permet de faire de très nombreuses choses avec un document XML : en extraire des informations, le reformater et même le modifier.

On ne présentera ici que les bases fondamentales, communes aux trois versions majeures de XQuery.

5.1. XQuery

5.1.1. Présentation

XQuery est un langage permettant de traiter un document XML. Comme XSLT, il produit un document en sortie. Les instructions XQuery se placent au milieu d'une sorte de modèle XML contenant ce qu'on veut. Le traitement consiste à remplacer les instructions par ce qu'elles calculent.

Les différences avec XSLT sont dans la syntaxe, XQuery n'est pas du XML². D'autre part la norme XPath utilisée est la version 2.0 qui offre des possibilités supplémentaires.

XQuery permet d'exprimer des requêtes beaucoup plus complexes que XPath version 1.0. Il permet de faire des sortes de jointures. De fait, XQuery est aux *bases de données XML* ce que SQL est aux SGBD relationnels.

5.1.2. Exemple initial

Pour un aperçu de ce qu'est XQuery, voici une feuille pour afficher [albums.xml](#) en HTML :



```
<html><body>
<h2>Albums de Tintin</h2>
<table>{
  for $album in doc("albums.xml")//album[@serie="Tintin"]
  return
    <tr>
      <td>{ $album/titre }</td>
      <td>{ concat($album//mois," ",$album//annee) }</td>
    </tr>
```

²Il existe une syntaxe XML mais très peu lisible.

```
}</table>  
</body></html>
```

5.1.3. Traitement d'une feuille XQuery

Pour lancer le traitement en ligne de commande, on peut employer un outil appelé [galax-run](#).

```
galax-run entrée > sortie
```

Par exemple : `galax-run albums.xq > albums.html`

Un autre logiciel, beaucoup plus puissant s'appelle [BaseX](#), décrit sur la page [wikipedia](#). Il permet de gérer une base de données XML et de l'interroger avec des requêtes XQuery. Voir la deuxième partie de ce cours.

5.1.4. Bases de XQuery

XPath est un sous-ensemble de XQuery. C'est à dire que XQuery exécute n'importe quelle requête XPath directement.

Exemple de requête XQuery XPath sur le document XML :



```
//album[@serie="Tintin" and date/mois="août"]/titre
```

Dans la suite, vous verrez qu'on met le mot clé `return` quand on veut générer quelque chose de différent du document d'origine. Il ne faut pas le mettre quand c'est une simple requête XPath.

5.1.5. Scripts XQuery

En général, une requête XQuery se place dans un fichier source `requete.xq`. Ce fichier contient, soit uniquement une requête, soit du code XML dans lequel il y a des requêtes placées entre `{...}`.

Exemple de source :



```
<html><body lang="fr">  
Il y a { count( doc("albums.xml")/albums/album ) } albums.  
</body></html>
```

- La fonction `doc("albums.xml")` retourne le document XML ; on lui applique une requête XPath.
- La fonction `count(collection)` compte le nombre de nœuds de la collection. Cette collection provient de l'expression XPath `/albums/album` appliquée au document `albums.xml`.

5.1.6. Génération d'éléments XML

Pour produire des éléments ou des attributs en sortie, il y a une syntaxe ressemblant à RelaxNG :

- `element NOM { CONTENU1, CONTENU2, ... }`
- `attribute NOM { VALEUR }`

Exemple de script XQuery qui retourne exactement le même résultat que l'exemple précédent : 

```
element html {  
  element body {  
    attribute lang { "fr" },  
    "Il y a",  
    count(doc("albums.xml")/albums/album),  
    "albums."  
  }  
}
```

5.1.7. Affectation de variables

XQuery permet de définir des variables. La syntaxe est :

```
let $NOM := VALEUR  
return SORTIE
```

- La valeur est une expression XPath. Notez le \$ devant les variables (comme en PHP).
- **return** indique ce qu'il faut produire en sortie.

Cet exemple est une variante des précédents :



```
let $nombre := count( doc("albums.xml")/albums/album )  
return <html><body>Il y a { $nombre } albums.</body></html>
```

Notez les {...} pour délimiter du code XQuery dans la clause **return**. Attention, il n'y a **pas de ;** à la fin du **let**.

5.1.8. Affectations multiples

On peut faire plusieurs affectations successives, liées ou non :



```
let $albums := doc("albums.xml")/albums/album  
let $nombre := count( $albums )  
let $min_annee := min( $albums/date/annee )  
let $max_annee := max( $albums/date/annee )  
return  
  element html {  
    element body {  
      "Il y a", $nombre, "albums de",  
      $min_annee, "à", $max_annee  
    }  
  }
```

Attention à ne pas mettre de ; dans cette requête. En fait, tout cela n'est qu'une seule instruction XQuery.

5.1.9. Conditionnelles

XPath 2.0 fournit une structure conditionnelle :

```
if (CONDITION) then EXPR1 else EXPR2
```

C'est une expression dont la valeur est soit EXPR1, soit EXPR2.

Exemple :

```
let $nombre := count( doc("albums.xml")/albums/album )
return <html><body>Il y a {
    if ($nombre > 20) then "de nombreux" else $nombre
} albums.</body></html>
```

Remarquez bien les mots clés **then** et **else** et ne les confondez pas avec les **{** des langages C et Java. Ici, les **{** et **}** permettent de passer de l'espace XML à l'espace XQuery.

5.1.10. Conditionnelles (suite)

Du fait que ce soit une expression, on **ne peut pas** l'employer ainsi :

```
let $nombre := count( doc("albums.xml")/albums/album )
if ($nombre > 20) then return "beaucoup" else return "peu"
```

Par contre, il y a une structure pour cela :

```
let AFFECTATION where (CONDITION) return SORTIE
```

Exemple :

```
let $nombre := count( doc("albums.xml")/albums/album )
where ($nombre > 20)
return <html><body>Il y a de nombreux albums.</body></html>
```

Cependant, il n'y a pas de clause **else** possible.

5.1.11. Boucles

La puissance de XQuery vient des boucles **for**. Le schéma général est appelé **FLWOR** (*For Let Where OrderBy Return*) (et non pas FLOWR), dont voici le plus simple (les clauses sont optionnelles) :

```
for VARIABLE in COLLECTION
return SORTIE
```

Exemple :

```
for $album in doc("albums.xml")/albums/album
  return element tr { element td { $album/titre/text() } }
```

La collection est générée par l'expression `/albums/album`. C'est la liste de tous les éléments `<album>` du document. Notez la fonction `text()` pour récupérer le contenu texte du titre.

5.1.12. Clause For

La clause `for` fait un parcours sur une collection :

- énumération d'entiers (INF to SUP), par exemple :

```
for $i in (1 to 10) return $i * $i
```

- requête XPath qui retourne une collection de nœuds XML :

```
let $albums := doc("albums.xml")//album
for $mois in distinct-values( $albums//mois )
  return element tr {
    count(//album[date/mois=$mois]), "en", $mois }
```

La fonction `distinct-values(collection)` retourne une collection sans doublons.

5.1.13. Clause For sur des attributs

Quand on veut itérer sur des attributs, comme dans :

```
for $attr_numero in doc("albums.xml")//album/@numero
```

La variable `$attr_numero` contient le nœud XML de type *attribute* et non pas seulement la valeur de l'attribut.

Il faut extraire la valeur comme ceci :

```
let $numero := string($attr_numero)
```

5.1.14. Clause Let


Il est possible d'insérer une ou plusieurs affectations avant et après la clause `for`, pour un calcul intermédiaire :

```
for VARIABLE1 in COLLECTION
  let VARIABLE2 := EXPRESSION
  return SORTIE
```

Exemple :

```
let $albums := doc("albums.xml")//album
for $mois in distinct-values( $albums//mois )
  let $nombre := count( $albums[date/mois=$mois] )
  let $titres := $albums[date/mois=$mois]/titre
  return element titres {
    attribute nombre { $nombre }, attribute mois { $mois },
    $titres
  }
```

5.1.15. Clause Where

C'est une condition optionnelle dans la boucle pour filtrer les itérations. Elle ressemble à la clause `where` des requêtes SQL. 

```
let $albums := doc("albums.xml")//album
for $album in $albums
  where $album/date/annee >= 1970
  return $album
```

Remarque: pour ça, on pourrait aussi écrire du XPath pur : 

```
doc("albums.xml")//album[date/annee>=1970]
```

5.1.16. Clause Order by

Cette clause optionnelle permet de classer les éléments à traiter dans la boucle. On peut rajouter `ascending` ou `descending` pour indiquer le sens.

```
for VARIABLE in COLLECTION
  order by EXPRESSION
  return SORTIE
```

Exemple : 

```
let $albums := doc("albums.xml")//album
for $album in $albums
  let $titre := $album/titre
  where starts-with($titre, "Tintin")
  order by $titre ascending
  return $titre
```

5.1.17. Boucles imbriquées

Exemple : 

```
let $albums := doc("albums.xml")//album
for $mois in distinct-values( $albums//mois )
  return element mois {
    attribute nom { $mois },
    for $album in $albums
      where $album/date/mois = $mois
      return $album/titre
  }
```

La clause **where** est une sorte de condition de jointure entre les mois et les albums.

La boucle interne pourrait être remplacée par une simple expression XPath.

5.2. Bases de données XML

5.2.1. Présentation

Une base de données XML native incorpore des données directement au format XML et propose les langages XPath et XQuery pour les interroger.

Ces bases de données sont des sortes de bases NoSQL : *not only SQL*. La structuration des données et les méthodes d'interrogation sont d'un autre genre que les requêtes SQL sur une base relationnelle.

Nous verrons une autre approche en fin de période, l'intégration de XML dans PostgreSQL.

5.2.2. Principe général d'un SGBD XML

Le SGBD stocke des « forêts » d'arbres XML qui peuvent provenir de différents documents XML ayant par exemple le même schéma.

Le SGBD fonctionne en mode client/serveur. Il exécute des requêtes de type XQuery à la demande des clients. Il existe des langages XML permettant de modifier les données, *XQuery Update*, et dans certains cas, le SGBD fournit un modèle REST³ pour les clients.

Les requêtes sur un gros document sont rendues efficaces à l'aide d'index sur les éléments.

5.2.3. Utilisation de BaseX

En TP, nous utiliserons [BaseX](#). Il y a également [eXist](#). Tous deux sont gratuits et open source. BaseX a été développé initialement par l'Université de Constance en Allemagne et il est maintenant sur GitHub, licence BSD.

Le SGBD [BaseX](#) propose :

- Interrogation à l'aide de XQuery (y compris version 3),
- Modification à l'aide de XQuery Update,
- Serveurs intégrés de type RESTful et WebDAV (mais il faut télécharger des archives spécifiques pour les avoir)
- Une interface utilisateur complète.

La documentation complète est disponible au format pdf sur [cette page](#).

³Un serveur REST propose plusieurs méthodes d'interrogation et de modifications à l'aide de requêtes HTTP GET, PUT, POST, DELETE. Chaque requête est complète et indépendante des autres.

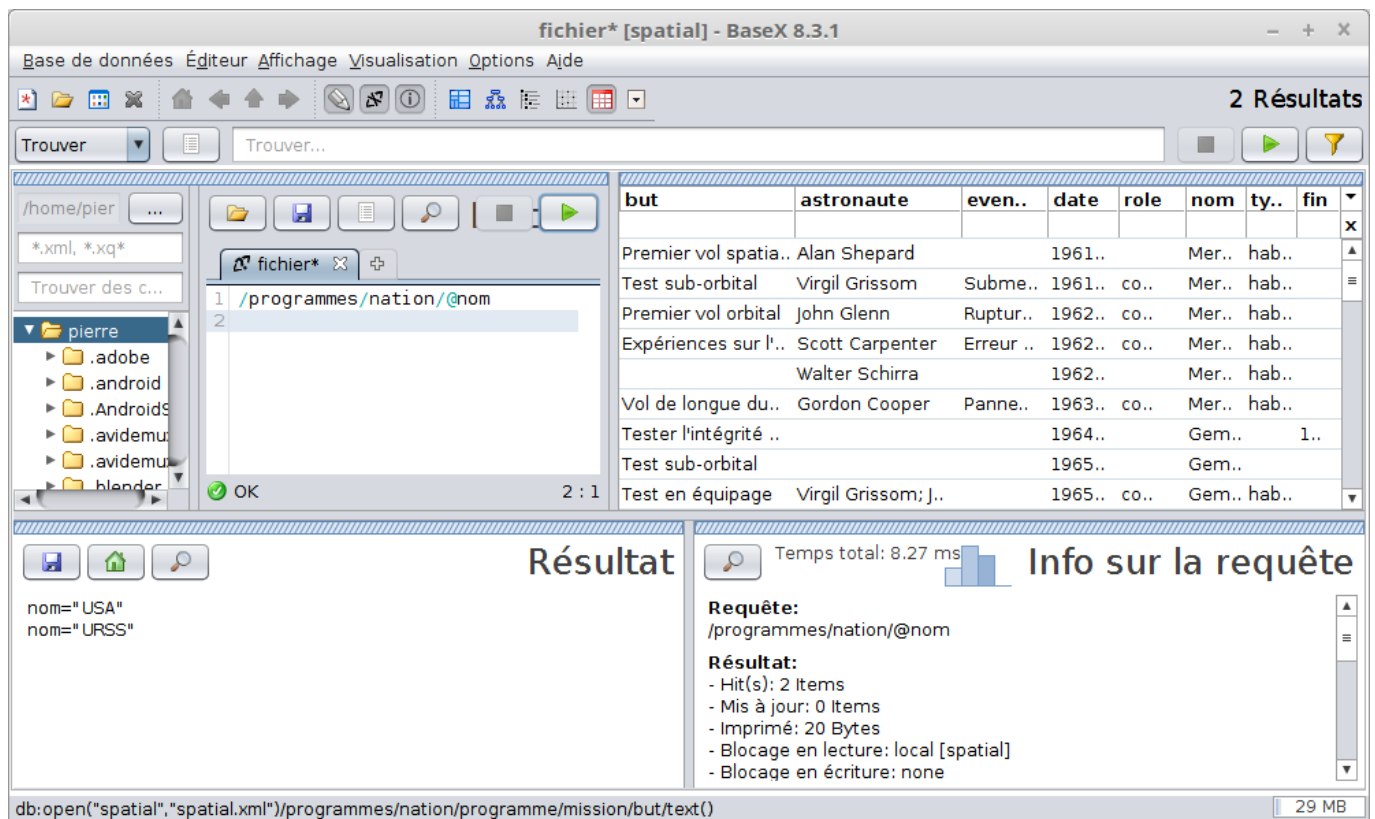


Figure 12: BaseX

5.2.4. Interface de BaseX

5.2.5. Interface graphique (suite)

L'interface est composée de plusieurs panneaux :

- Une barre pour saisir directement des recherches texte ou des requêtes XQuery ou des commandes sur la base,
- Un éditeur de requête XQuery,
- Une vue tabulaire de la base XML,
- Les résultats de la requête, qu'on peut aussi afficher graphiquement selon le type des données,
- Des statistiques sur l'exécution de la requête.

Elle est documentée sur [cette page](#).

En fait, c'est tellement complet qu'on ne pourra faire que quelques manipulations de découverte en TP.

5.2.6. Création d'une base de données XML

Il faut posséder un fichier de données au format XML. C'est une collection de données toutes similaires, par exemple comme la liste des albums de Tintin ou un annuaire téléphonique. Il faut que ces données puissent être distinguées entre elles, par exemple par un attribut identifiant.

Soit avec les menus, soit avec une commande (voir en TP), on fait prendre en charge ce fichier par BaseX.

but	astronaute	evenement	date	role	nom	type	fin
Premier vol spatial américain	Alan Shepard		1961-05-05		Mercury 3	habité	
Test sub-orbital	Virgil Grissom	Submersion de..	1961-06-21	command..	Mercury 4	habité	
Premier vol orbital	John Glenn	Rupture de la f..	1962-02-20	command..	Mercury 6	habité	
Expériences sur l'impesanteur	Scott Carpenter	Erreur de pilot..	1962-05-24	command..	Mercury 7	habité	
	Walter Schirra		1962-10-03		Mercury 8	habité	
Vol de longue durée	Gordon Cooper	Panne électriq..	1963-05-15	command..	Mercury 9	habité	
Tester l'intégrité de la capsule			1964-04-08		Gemini 1		1964-..
Test sub-orbital			1965-01-19		Gemini 2		
Test en équipage	Virgil Grissom; John Young		1965-03-23	command..	Gemini 3	habité	
Sortie dans l'espace	James McDivitt; Edward White	Blocage du lo..	1965-06-03	command..	Gemini 4	habité	1965-..
Vol de longue durée	Gordon Cooper; Pete Conrad	Panne d'une pi..	1965-08-21	command..	Gemini 5	habité	1965-..
Rendez-vous orbital	Walter Schirra; Thomas Stafford		1965-12-15		Gemini 6	habité	1965-..
Vol de longue durée	Franck Borman; Jim Lovell		1965-12-04	command..	Gemini 7	habité	1965-..
Rendez-vous orbital	Neil Armstrong; Dave Scott	Rotation incont..	1966-03-16	command..	Gemini 8	habité	1966-..
Rendez-vous orbital; Sortie dans l..	Thomas Stafford; Gene Cernan	Impossibilité d..	1966-06-03	command..	Gemini 9	habité	1966-..
Rendez-vous orbital; Sortie dans l..	John Young; Michael Collins		1966-07-18	command..	Gemini 10	habité	1966-..
Rendez-vous orbital	Pete Conrad; Richard Gordon		1966-09-12	command..	Gemini 11	habité	1966-..
Rendez-vous orbital; Sortie dans l..	Jim Lovell; Edwin Aldrin		1966-11-11	command..	Gemini 12	habité	1966-..

Figure 13: Table XML

Voir la figure 13, page 82.

5.2.7. Requête XQuery

On peut ensuite écrire des requêtes XPath ou XQuery dans la barre ou dans la zone d'édition et lancer l'exécution.

Voir la figure 14, page 83.

5.3. XQuery pour modifier la base XML

5.3.1. Requêtes en modification

Il existe un langage appelé XQuery Update Facility (XQUF), documenté sur [cette page](#), qui permet de modifier les données XML de la base. Il ajoute de nouvelles requêtes parmi lesquelles :

- **insert node chose into expression** : chose décrit un nouveau node (élément ou attribut) qui doit être inséré dans les données aux emplacements désignés par l'expression.
- **delete node expression** : supprime l'élément ou l'attribut désigné par l'expression.
- **replace [value of] node expression with chose** : remplace le nœud (ou sa valeur) désigné avec l'expression par la chose.

Ces requêtes modifient des nodes, ce sont toutes sortes de nœuds dans l'arbre XML concerné : éléments, attributs, textes, commentaires, CDATA, etc.

5.3.2. Insertion d'éléments

On peut écrire l'élément à ajouter en syntaxe XML ou avec la syntaxe XQuery `element nom { contenu }`.

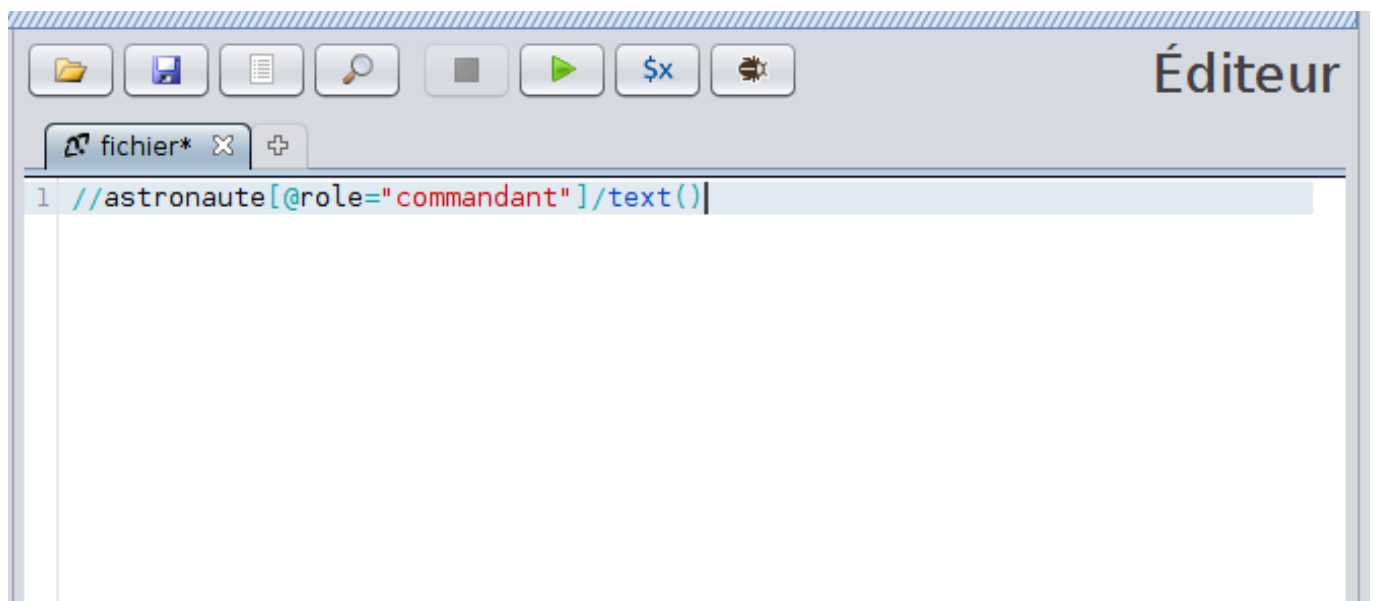


Figure 14: Requête XPath

Exemple, on rajoute un nouvel album dans [albums.xml](#) et un sous-élément `<lu/>` dans l'album n°1 :



```
insert node
  <album numero="25">
    <titre>Tintin et Astérix contre Spirou</titre>
  </album>
into /albums
insert node element lu {} into /albums/album[@numero=1]
```

Important : l'expression après le `into` doit désigner un élément unique.

5.3.3. Insertion sur plusieurs éléments

Pour systématiser l'insertion sur plusieurs éléments, il faut simplement utiliser une expression FLWOR. Voici un exemple, on veut ajouter l'élément `<achat date="2019-12-24"/>` dans tous les albums à partir du n°10 :



```
for $album in /albums/album
  where $album/@numero >= 10
  return insert node <achat date="2019-12-24"/> into $album
```

L'astuce est de mettre le `insert` en tant que `return`.

5.3.4. Insertion d'attributs

Il suffit de l'écrire à l'aide de la syntaxe XQuery `attribute nom {'valeur'}`.

Exemple, on rajoute un attribut `editeur` pour l'album n°13 :



```
insert node attribute editeur {'Casterman'}  
into /albums/album[@numero="13"]
```

Important : comme précédemment, l'expression `into` doit désigner un emplacement unique, mais on peut utiliser une boucle pour insérer le même attribut dans des éléments différents.

5.3.5. Suppression d'éléments ou d'attributs

Voici trois exemples, on supprime l'album n°4, puis tous les albums parus en janvier, et enfin l'attribut `numero` des albums parus après 1950 (ça casse les données !) :

```
delete node /albums/album[@numero="4"]  
delete node /albums/album[date/mois="janvier"]  
delete node /albums/album[date/annee>1950]/@numero
```

Contrairement aux `insert`, `replace` et `rename`, un `delete` peut concerner plusieurs nodes.

5.3.6. Remplacement d'éléments ou d'attributs

Voici un exemple, on remplace le contenu de l'élément `<titre>` de l'album n°1 :

```
replace value of node /albums/album[@numero=1]/titre  
with 'nouveau titre'
```

Dans cet exemple, son attribut `numero` devient -1 :

```
replace value of node /albums/album[@numero=1]/@numero  
with -1
```

5.3.7. Remplacement d'éléments ou d'attributs

On peut aussi remplacer un élément par autre chose. Par exemple, remplacer l'élément `<date>` et ses descendants par tout autre chose (ça peut casser le schéma) :

```
replace node /albums/album[@numero=2]/date  
with <auteur nom="hergé"/>
```

Il existe aussi une requête pour renommer un élément ou un attribut (faire une boucle s'il y en a plusieurs) :

```
rename node designation as nouveau nom
```

5.3.8. Autres actions

Il y a de nombreuses autres actions possibles, soit des raffinements des actions comme `insert`, soit d'autres opérations plus spécifiques pour faire des sortes de transactions. Elles sont trop complexes pour être présentées ici et ne seront pas nécessaires en TP.

Semaine 6

API W3C DOM

Le cours de cette semaine présente l'API XML DOM permettant de produire et traiter un document XML :

- Principes,
- Création et modification d'un XML,
- Lecture et traitement d'un XML,
- Sérialisation d'objets Java en XML,
- XML DOM dans d'autres langages.

6.1. Principes

6.1.1. Présentation

Une [interface de programmation](#) (*Application Programming Interface* API en anglais) est un ensemble de bibliothèques de fonctions et d'outils permettant d'écrire des programmes spécialisés.

L'API DOM est définie par le [W3C](#), c'est à dire le *World Wide Web Consortium* qui normalise tout ce qui concerne le Web, dont XML.

Le sigle [DOM](#) signifie *Document Object Model*. Cette API manipule une représentation d'un document complet. La totalité du document est chargée en mémoire pendant le traitement.

Il existe une autre API appelée [SAX](#) (*Simple API for XML*) qui permet de lire un document XML de manière séquentielle sans rien stocker en mémoire. Voir le prochain cours.

6.1.2. Principe généraux de l'API DOM

L'API W3C DOM se programme avec un langage objet : Java, JavaScript, PHP, Python, C++... Quand on crée ou qu'on ouvre un document XML, ça crée une instance qui représente le document tout entier. Ensuite, on utilise les méthodes de cette instance pour créer ou parcourir les éléments, attributs et textes du document.

- En mode création :
 1. créer une instance de `Document`,
 2. ajouter des instances d'`Element` au document,
 - a. leur ajouter des attributs, textes, CDATA...
 3. écrire le document dans un fichier ou sur le réseau.
- En mode lecture d'un fichier :
 1. créer une instance de `Document`,
 2. ouvrir et analyser un fichier XML, ça remplit le document,
 3. parcourir les instances d'`Element` du document.

6.1.3. Bibliothèques

Pour travailler avec l'API, il faut importer un petit nombre de librairies :



```
import java.io.File;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;
```

6.2. Document DOM en mode création

6.2.1. Création d'un Document

En Java, il faut trois instructions :



```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();
```

1. Création d'une factory : c'est un singleton qui permet de créer des objets d'un certain type, ici des DocumentBuilder.
2. Création d'un builder : encore un singleton mais spécialisé dans la création de documents XML.
3. Création d'un document : c'est lui qui représente le document XML qu'on veut manipuler.

6.2.2. Compléments

Le code complet se présente comme ceci :




```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;

void CreationXML()
{
    try {
        DocumentBuilderFactory factory = DBF....newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
```

```
Document document = builder.newDocument();  
...  
} catch (Exception e) {...}  
}
```

6.2.3. Création d'éléments

La classe `Document` possède des méthodes pour rajouter des éléments. Ça se passe en deux temps :

1. Création d'un élément : `document.createElement(nom);`
2. Ajout de cet élément dans le document, en tant qu'enfant d'un élément existant :
`parent.appendChild(enfant);` 

```
import org.w3c.dom.Element;  
  
// création de la racine du document  
Element racine = document.createElement("voiture");  
document.appendChild(racine);  
// ajout d'un élément sous la racine  
Element marque = document.createElement("marque");  
racine.appendChild(marque);
```

6.2.4. Insertion avant un autre élément

La méthode `parent.appendChild(element)` ajoute l'élément à la fin de la liste du parent.

On peut insérer un élément avant un autre avec : 

```
Element prix = document.createElement("prix");  
racine.insertBefore(prix, marque);
```

Cela va insérer `<prix>` avant `<marque>` dans la racine.

NB: le nœud parent d'un élément s'obtient par : 


```
Element parent = (Element) element.getParentNode();
```

6.2.5. Création d'un arbre d'éléments

On pourrait créer des éléments à la volée de cette manière : 


```
// ajout de plusieurs éléments sous la racine  
racine.appendChild(document.createElement("marque"));  
racine.appendChild(document.createElement("couleur"));
```

Mais on a aucune variable pour représenter les éléments rajoutés, on ne peut pas leur rajouter des enfants et des attributs.

Pour créer un arbre complexe, il faut définir des variables pour chacun des éléments. Cela peut passer par des tableaux : 


```
Element[] annees = new Element[4];
for (int i=0; i<4; i++) {
    annees[i] = document.createElement("annee");
    racine.appendChild(annees[i]);
}
```

6.2.6. Ajout d'attributs aux éléments

Placer des attributs sur un élément est très facile. On peut manipuler l'attribut en tant qu'objet : 

```
import org.w3c.dom.Attr;

Attr attribut = document.createAttribute("attribut");
attribut.setValue("valeur");
element.setAttributeNode(attribut);
```

ou plus simplement : 

```
element.setAttribute("attribut", "valeur");
```

Notez que les noms et valeurs sont des chaînes. Si vous avez des nombres à affecter, il faudra les convertir en textes avec `String.valueOf(nombre)`.

6.2.7. Espaces de nommage

Lorsqu'un élément doit avoir un *namespace* identifié par un *URI* et un préfixe, il faut créer l'élément avec la méthode `createElementNS(URI, nom qualifié)` :


Rappel du cours 1 : le nom qualifié est composé d'un préfixe et d'un nom local séparés par : 

```
final static String URI = "urn:iutlan:test";
final static String PREFIXE = "iutlan:";
Element element =
    document.createElementNS(URI, PREFIXE+"element");
```

De même avec les attributs : 

```
element.setAttributeNS(URI, PREFIXE+"attribut", "valeur");
```

6.2.8. Ajout de textes et de CDATA

Nous arrivons au contenu d'un élément. Il est très simple de rajouter du texte dans un élément. Il n'est pas forcément nécessaire d'associer une variable sauf si le texte doit être modifié ultérieurement. Il y a deux possibilités, la seconde est plus simple : 


```
element.appendChild(document.createTextNode("texte"));  
element.setTextContent("texte");
```

On peut aussi rajouter des sections CDATA par :

```
element.appendChild(document.createCDATASection("data"));
```

NB: les sections CDATA sont des nœuds frères des textes et non pas des nœuds enfants.

6.2.9. Ajout de commentaires et autres

C'est aussi simple que de rajouter du texte :

```
element.appendChild(document.createComment("commentaire"));
```

Il y a également une méthode pour créer une instruction de traitement, c'est à dire un nœud `<?action parametres?>` :

```
racine.appendChild(  
    document.createProcessingInstruction(  
        "xml-stylesheet", "href='style.xsl' type='text/xsl'"));
```

Chaque sorte de *Node* peut être ajoutée au document de cette manière.

6.2.10. Enregistrement dans un fichier

C'est à faire tout à la fin, lorsque le document est complet.

```
import javax.xml.transform.Transformer;  
import javax.xml.transform.TransformerFactory;  
import javax.xml.transform.dom.DOMSource;  
import javax.xml.transform.stream.StreamResult;  
  
// écrivain  
TransformerFactory transformerFactory =  
    TransformerFactory.newInstance();  
Transformer transformer = transformerFactory.newTransformer();  
  
// écriture du document dans un fichier  
DOMSource source = new DOMSource(document);  
StreamResult sortie = new StreamResult(new File("sortie.xml"));  
transformer.transform(source, sortie);
```

6.3. Document DOM en mode lecture

6.3.1. Traitement du document

On se place maintenant du côté lecture et analyse d'un document XML existant. En général, on a besoin de :

- chercher un ou plusieurs nœuds spécifiques,
- itérer sur tous les nœuds enfants d'un nœud,
- vérifier le nom d'un nœud,
- extraire les valeurs d'attributs ou le contenu texte d'un nœud.

C'est souvent un ensemble de tout cela.

6.3.2. Ouverture d'un fichier

Pour ouvrir un fichier XML existant, le début est similaire à la création d'un document :



```
DocumentBuilderFactory factory =  
    DocumentBuilderFactory.newInstance();  
factory.setNamespaceAware(true);  
DocumentBuilder builder = factory.newDocumentBuilder();  
Document document = builder.parse(new File("document.xml"));
```

Notez les deux changements :

- (ligne 2) On prévient qu'il va y avoir des *namespaces*,
- (ligne 4) On remplit le document avec ce qui se trouve dans le fichier XML.

En fait, on peut aussi compléter ou modifier le document existant puis l'enregistrer comme dans la partie précédente.

6.3.3. Classe Node

Les classes `Element`, `TextNode`, `Comment`... sont toutes des sous-classes de `Node`⁴. Un `Node` représente l'un des nœuds de l'arbre XML sous-jacent (voir cours 1).

Dans le modèle W3C, un `Node` possède un *type*. C'est un petit entier `short` retourné par la méthode `getNodeType()`. Des constantes permettent de nommer ces types :

`Node.ELEMENT_NODE` pour les `Node` de type `Element`
`Node.TEXT_NODE` pour les `Node` de type `Text`
`Node.DOCUMENT_NODE` pour les `Node` de type `Document`
`Node.ATTRIBUTE_NODE` pour les `Node` de type `Attr`
`Node.COMMENT_NODE` pour les `Node` de type `Comment`

6.3.4. Modification d'un document

Les méthodes suivantes permettent de modifier un document :

⁴En réalité, en Java, ce sont des interfaces et non pas des classes.

- `element.appendChild(node)` pour ajouter le nœud (élément, texte...) après tous les enfants de l'élément.
- `element.insertBefore(node, autre)` ajoute le nœud avant *autre* parmi les enfants de l'élément.
- `element.removeChild(node)` retire le nœud indiqué de la liste de l'élément.
- `document.renameNode(node, URI, nom qualifié)` change le nom du nœud indiqué. Mettre URI à null s'il n'y a pas de *namespace*.

6.3.5. Prologue du document

Des méthodes de `Document` permettent d'obtenir les informations du prologue :

- `String document.xmlVersion()` retourne la version, c'est "1.0" en général.
- `String document.xmlEncoding()` retourne l'encodage, par exemple "UTF-8".

NB: il n'est pas du tout nécessaire de récupérer ces informations pour traiter le document.

Il faut également noter que les *setters* existent pour configurer un document en création/modification. Par exemple `document.setXmlVersion("1.1");`

6.3.6. Élément racine

On obtient l'objet Java représentant la racine du document par :



```
Element racine = document.getDocumentElement();
```

NB: cet élément est unique, sinon le fichier XML est mal formé.

C'est une instance de la classe `Element`. Puis pour avoir le nom de la racine, on emploie l'un de ses *getters* :



```
String nom = racine.getNodeName();
```

Dans un programme, on se contente en général de vérifier que la racine porte le bon nom.

6.3.7. Espaces de nommages

Le nom d'un élément s'obtient par `getNodeName()` ou `getTagName()` qui est équivalente.

Lorsqu'il y a un *namespace*, le nom de l'élément s'appelle un *nom qualifié*, c'est ce qui est retourné par les deux méthodes précédentes, et il est composé d'un *préfixe* séparé du *nom local* par un « : »

On peut obtenir :

- le préfixe : `String element.getPrefix()`
- le nom local : `String element.getLocalName()`
- l'URI du préfixe : `String element.getNamespaceURI()`

NB: toutes ces méthodes renvoient null si on a oublié de mettre `factory.setNamespaceAware(true);` avant de charger le fichier.

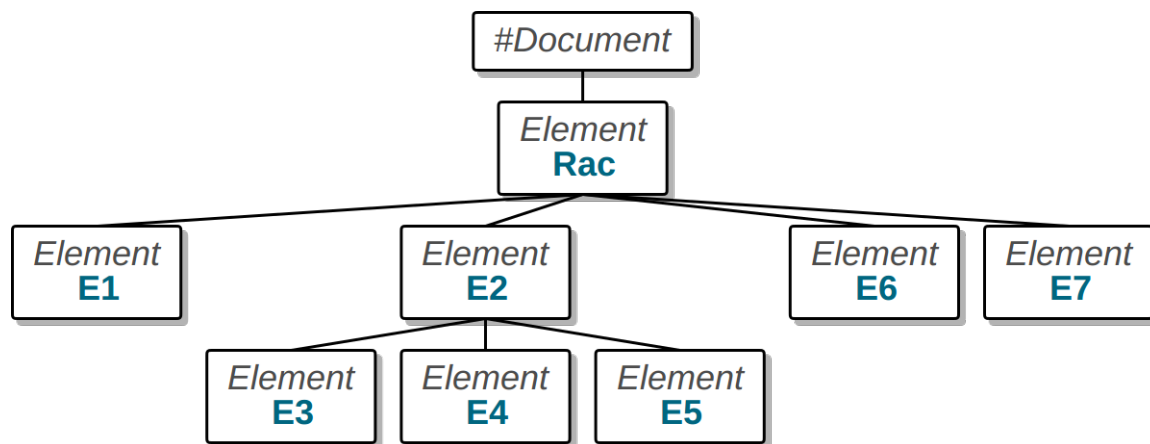


Figure 15: Arbre XML

6.3.8. Attributs d'un Element

Les méthodes suivantes permettent d'obtenir les attributs d'un élément :

- `String element.getAttribute(nomattr)` retourne l'attribut ou la chaîne vide s'il n'y a pas cet attribut. C'est pour distinguer la présence d'un attribut qui serait vide de son absence qu'il faut tester auparavant avec la méthode suivante,
- `boolean element.hasAttribute(nomattr)` renvoie `true` si l'élément possède cet attribut

Il y a des méthodes pour tenir compte des *namespaces* des attributs. Il faut leur fournir l'URI qui définit le *namespace* :

- `String element.getAttributeNS(URI,nomlocal)`
- `boolean element.hasAttributeNS(URI,nomlocal)`

6.3.9. Nœuds enfants d'un élément

Une instance de la classe `Element` telle que la racine du document peut avoir un `Node` parent, des enfants, ainsi que des frères. Voici un schéma pour le transparent suivant :

6.3.10. Voisinage d'un nœud

Quand on considère le nœud E2 :

- Le nœud parent de E2 est Rac
 - On l'obtient par `E2.getParentNode()`
- Le précédent nœud frère de E2 est E1
 - On l'obtient par `E2.getPreviousSibling()`
- Le nœud frère suivant est E6
 - On l'obtient par `E2.getNextSibling()`
- Le premier nœud enfant de E2 est E3
 - On l'obtient par `E2.getFirstChild()`
- Le dernier nœud enfant de E2 est E5
 - On l'obtient par `E2.getLastChild()`

Toutes ces méthodes retournent `null` si aucun `Node` ne correspond.

6.3.11. Parcours des nœuds enfants (méthode 1)

Pour passer les enfants d'un élément en revue, on peut utiliser l'algorithme suivant :



```
Node courant = element.getFirstChild();
while (courant != null) {
    // traiter le noeud courant
    ...
    // passer au suivant
    courant = courant.getNextSibling();
}
```

6.3.12. Parcours des nœuds enfants (méthode 2)

On peut aussi utiliser la méthode `getChildNodes()` qui retourne une liste de `Node` dans un objet de type `NodeList`. C'est une sorte de tableau dont on peut récupérer la taille et l'un des éléments, un `Node`, par son indice. Voici l'algorithme :



```
NodeList liste = element.getChildNodes();
final int nombre = liste.getLength();
for (int i=0; i<nombre; i++) {
    Node courant = liste.item(i);
    // traiter le noeud courant
    ...
}
```

Le mot clé Java `final` signifie que la variable ne changera plus après son affectation. Ça accélère un peu les boucles.

6.3.13. Parcours des nœuds enfants (méthode 3)

Il y a encore une autre manière de parcourir certains enfants d'un élément, en utilisant la méthode `getElementsByTagName(nom)` qui retourne une `NodeList` des éléments ayant le nom indiqué.



```
NodeList liste = element.getElementsByTagName("voiture");
final int nombre = liste.getLength();
for (int i=0; i<nombre; i++) {
    Node courant = liste.item(i);
    // traiter le noeud courant
    ...
}
```

Il y a une variante avec `namespace` : `getElementsByTagNameNS`

6.3.14. Parcours des nœuds enfants (méthode 4)

Il existe enfin une 4e manière pour trouver directement les éléments qu'on souhaite dans un document XML. Elle est basée sur l'attribut spécial `xml:id` (de type ID dans une DTD).

```
<voiture xml:id="voiture1">...</voiture>
<voiture xml:id="voiture2">...</voiture>
```

La méthode `getElementById("code")` de la classe `Document` trouve l'élément portant l'attribut `xml:id="code"` ou `null` s'il n'y en a pas dans le document.

Par exemple, on cherche la voiture2 :

```
Element voiture2 = document.getElementById("voiture2");
```

On peut ensuite directement traiter l'élément (sauf si `null`).

6.3.15. Traitement d'un nœud

On étudie maintenant ce qui est fait dans le cœur de la boucle des algorithmes précédents (méthodes 1 à 3).

D'abord faire attention, ce ne sont pas forcément que des instances d'`Element`, ça peut être des commentaires, des textes ou d'autres nœuds. Il faut donc faire un test sur le type de nœud :

```
Node courant = ...
// traiter le nœud courant
switch (courant.getNodeType()) {
    case Node.ELEMENT_NODE:    // c'est un élément
        break;
    case Node.TEXT_NODE:       // c'est un texte
        break;
    case Node.COMMENT_NODE:    // c'est un commentaire
        break;
    ...
}
```

6.3.16. Traitement d'un élément

Dans la pratique, on se contente des tests qui nous intéressent afin d'extraire les données dont on a besoin. Par exemple :

```
Node courant = ...
// traiter le nœud courant
if (courant.getNodeType() == Node.ELEMENT_NODE &&
    courant.getNodeName().equals("voiture")) {
    // on est sur un élément <voiture>
    Element voiture = (Element) courant;
    // traiter cet élément
    ...
}
```

La conversion du `Node` en `Element` permet d'utiliser les *getters* spécifiques pour avoir ses attributs ou son contenu.

6.3.17. Traitement d'éléments

Quand différents éléments peuvent être mélangés et arriver dans n'importe quel ordre, il faut faire une sorte d'aiguillage. Java 1.7 permet de faire une conditionnelle multiple basée sur des chaînes, ici sur le nom de l'élément courant :

```
if (courant.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) courant;
    switch (element.getNodeName()) {
        case "voiture":
            ...
            break;
        case "prix":
            ...
            break;
    }
}
```

6.3.18. Contenu d'un nœud texte

Soit un `Element` représentant la marque de la voiture, correspondant à `<marque>Renault</marque>`. Comment faire pour récupérer le contenu texte, "Renault" de cet élément ?

On utilise la méthode `getTextContent()` :

```
// on est sur un élément <marque>
Element marque = (Element) courant;
String texte = marque.getTextContent();
```

Important: il faut savoir que `getTextContent()` concatène tous les textes contenus dans l'élément et tous ses sous-éléments, y compris les sections CDATA et les entités remplacées par leurs valeurs.

6.4. Sérialisation d'objets Java

6.4.1. Principes

La sérialisation d'un objet consiste à écrire ses variables membres dans un format pouvant être transmis ou stocké, et relu pour recréer les objets à l'identique. Nous allons voir comment générer un document XML à partir d'un objet Java et inversement, instancier un objet Java par un document XML.

En anglais, la sérialisation se dit *marshalling*, voir [wikipedia](#). Mais en Java, la *sérialisation* signifie plutôt l'extraction des données binaires présentes dans la mémoire pour représenter un objet.

Inversement, la dé-sérialisation (*unmarshalling*) consiste à instancier un objet à partir de ce qu'on lit dans le document : on reconstruit un objet à partir d'un document XML.

Pour cela, on utilise l'API DOM . Les variables membres sont écrites ou relues dans les sous-éléments ou attributs.

6.4.2. Exemple

Voici une classe :

```
class Voiture
{
    private int id;
    private String modele;
    private int kilometrage;
}
```

Une sérialisation XML d'une instance pourrait être :

```
<voiture id="1">
    <modele>Porsche 911</modele>
    <kilometrage>124999</kilometrage>
</voiture>
```

6.4.3. Récursivité

Lorsqu'une variable membre est un objet, lui-aussi est sérialisé :

```
class Voiture {
    private int id;
    private String modele;
    private Achat achat;
}
class Achat {
    private Date date;
    private float prix;
}
```

```
<voiture id="1">
    <modele>Porsche 911</modele>
    <achat date="2020-02-03">
        <prix>124999.90</prix>
    </achat>
</voiture>
```

6.4.4. Sérialisation manuelle

Principe : une méthode ressemblant à toString :

```
public void toXML(Document document, Node parent)
{
    // sérialisation de this dans le document
    // à l'intérieur de l'élément parent
}
```


Elle reçoit deux paramètres :

- un document XML représentant le fichier XML en construction,
- un Node parent afin d'accrocher ce qu'on va produire.

Comme `toString`, elle va écrire chaque variable membre dans le document.

- On crée d'abord l'élément racine du même nom que la classe :

```
Element elClasse = document.createElement("classe");  
parent.appendChild(elClasse);
```

- On ajoute les attributs convertis en chaînes :

```
elClasse.setAttribute("id", Integer.toString(id));
```

- On ajoute les sous-éléments :

```
Element elMembre = document.createElement("membre");  
elMembre.appendChild(document.createTextNode(membre));  
elClasse.appendChild(elMembre);
```

- Si une variable membre est un objet, alors on fait un appel récursif à `toXML(document, elClasse)`

6.4.5. Exemple de sérialisation

Voici l'exemple avec la classe Voiture précédente :



```
class Voiture {  
    public void toXML(Document document, Node parent) {  
        // élément <voiture>  
        Element elVoiture = document.createElement("voiture");  
        parent.appendChild(elVoiture);  
        // membre id sous la forme d'un attribut  
        elVoiture.setAttribute("id", Integer.toString(id));  
        // membre modele sous la forme d'un élément  
        Element elModele = document.createElement("modele");  
        elModele.appendChild(document.createTextNode(modele));  
        elVoiture.appendChild(elModele);  
        // membre achat par un appel récursif  
        achat.toXML(document, elVoiture);  
    }  
}
```

6.4.6. Cas des collections

Lorsqu'une variable membre est une collection, il faut ajouter une boucle pour écrire ses éléments. En général, on place les éléments de la collection dans un élément XML les regroupant. Exemple :

```
class Client {  
    private int id;  
    private List<Voiture> voitures;  
}
```

```
<client id="1">  
    <voitures>  
        <voiture id="1">...</voiture>  
        <voiture id="3">...</voiture>  
        <voiture id="7">...</voiture>  
    </voitures>  
</client>
```

6.4.7. Principes de désérialisation manuelle

La dé-sérialisation consiste à relire un élément XML pour en faire une instance de classe. C'est donc une méthode de type *fabrique* qui effectue cette opération.

```
public static Voiture fromXML(Node node) throws Exception  
{  
    // lire les elements, attributs du node  
    ...  
  
    // créer une instance si tout est ok  
    return new Voiture(id, modele, achat);  
}
```

Dans le cas général, c'est nettement plus compliqué que la sérialisation. Déjà, il faut vérifier la validité du document afin de ne pas rencontrer d'élément inconnu ou mal placé.

- Il faut commencer par vérifier que le node est un élément du bon type :

```
// vérifier que node est un élément <voiture>  
if (node.getNodeType() != Node.ELEMENT_NODE)  
    throw new Exception("node n'est pas un élément");  
if (! node.getNodeName().equals("voiture"))  
    throw new Exception("node n'est pas un élément voiture");
```

- Il est pratique de convertir le Node en Element :

```
Element elVoiture = (Element) node;
```

- Les membres qui sont des attributs peuvent être lus facilement

```
int id = Integer.parseInt(elVoiture.getAttribute("id"));
```

- Les membres qui sont des sous-éléments sont plus difficiles à capturer. Il faut parcourir les Node enfants, voir page 93 :

```
Node enfant = elVoiture.getFirstChild();
while (enfant != null) {
    if (enfant.getNodeType() == Node.ELEMENT_NODE) {
        switch (enfant.getNodeName()) {
            case "modele":
                modele = enfant.getTextContent();
                break;
            case "achat":
                achat = Achat.fromXML(enfant);
                break;
        }
    }
    enfant = enfant.getNextSibling();
}
```

6.5. Séri­a­li­sa­tion avec des outils

6.5.1. Présen­ta­tion

Afin de ne pas réinventer la roue, la sérialisation se fait de préférence avec une API (jars et outils) existante. Il y a deux catégories :

1. Les API basées sur des annotations simples :

Le principe est d'ajouter des annotations dans les classes à sérialiser. On désigne ce qui deviendra un élément et ce qui deviendra un attribut. Dans certains cas, l'API est capable de trouver elle-même les choses à sérialiser.

- [JAXB](#) était l'outil standard pour sérialiser des objets Java en XML, mais semble abandonné depuis Java 9.
- [XStream](#), est le plus simple et direct, aucune modification requise,
- [Simple](#) à peine plus compliqué.

2. Les API basées sur un document de liaison XML-Java :

Le principe est de créer un document ressemblant à un schéma de validation XSD et indiquant comment chaque membre de classe est sérialisé en XML. C'est beaucoup plus polyvalent que les solutions précédentes, mais la construction du document de liaison peut être difficile sans outil.

- [JiBX](#) serait intéressant mais il n'a pas encore été mis à jour pour Java 9+, il ne se compile pas.
- [XMLBeans](#) semble être le meilleur (maintenu, open source, performant), mais il est trop complexe pour le peu de temps disponible pour ce cours.

6.6. Séri­a­li­sa­tion avec l'API XStream

6.6.1. Présen­ta­tion

Cette API est ultra-simple, vous n'avez rien à modifier dans vos classes, sauf vous désirez un résultat spécifique. Les variables membres sont automatiquement enregistrées ou relues dans un document XML. Il suffit seulement de programmer la lecture ou l'écriture du document XML.

6.6.2. Sérialisation avec XStream

Prenez une classe Java quelconque, par exemple Voiture. Sa méthode `toXML` s'écrit ainsi : 

```
public void toXML(String filename) throws Exception
{
    // voir plus loin
    XStream xstream = createXStream();

    // sérialisation
    File output = new File(filename);
    FileWriter writer = new FileWriter(output);
    xstream.toXML(voiture, writer);
}
```

Pour la lancer :

```
maVoiture.toXML("voiture.xml");
```

6.6.3. Désérialisation avec XStream

Pour relire le fichier XML et instancier une Voiture : 

```
public static Voiture fromXML(String filename) throws Exception
{
    // voir plus loin
    XStream xstream = createXStream();

    // désérialisation
    File input = new File(filename);
    FileReader reader = new FileReader(input);
    return (Voiture) xstream.fromXML(reader);
}
```

Pour la lancer :

```
Voiture maVoiture = Voiture.fromXML("voiture.xml");
```

6.6.4. Objet XStream

L'instance `XStream xstream` est au cœur du processus. Il faut la configurer. C'était très simple en Java 8, mais Java 9+ pose des restrictions très fortes sur les techniques d'introspection (ou réflexivité), c'est à dire le fait pour un programme Java d'explorer son propre code exécutable. C'est l'introspection qui permet à XStream de trouver les noms et types des variables membres à sérialiser. Java 9 bloque cette inspection pour des raisons de sécurité. Seules des classes déclarées explicitement comme « inspectables » peuvent être explorées.

On arrive donc à une initialisation assez complexe de cet objet `XStream` en Java 9+. Le fichier [UtilXML.java](#) contient le code complet.

Voici un résumé :

```
public static XStream createXStream() {  
    // instance de XStream  
    XStream xstream = new XStream(new StaxDriver(){OUT}) {CFG};  
    xstream.autodetectAnnotations(true);  
  
    // permissions, ici toutes les classes sont acceptées  
    xstream.addPermission(AnyTypePermission.ANY);  
    return xstream;  
}
```

Les blocs notés `{OUT}` et `{CFG}` servent respectivement à configurer la production du XML en sortie et à configurer les inspections. Voir [UtilXML.java](#) pour le détail, parce que c'est très complexe, voir [xstream issue 101](#).

6.6.5. Ajustement de la sortie XML

`XStream` donne des noms basés sur les noms des classes aux éléments. Cela peut être déplaisant. Voici plusieurs possibilités de configuration :

- changer le nom de l'élément représentant une classe : `xstream.alias("NOM", CLASSE.class);`
- mettre une variable membre en attribut : `xstream.useAttributeFor(CLASSE.class, "MEMBRE");`
- mettre une collection à plat au lieu de l'imbriquer dans un élément racine : `xstream.addImplicitCollection(CLASSE.class, "MEMBRE");`

Par exemple, pour les voitures possédées par un client :

```
class Client {  
    private int id;  
    private List<Voiture> voitures;  
}
```

```
xstream.useAttributeFor(Client.class, "id");  
xstream.alias("client", Client.class);  
xstream.alias("voiture", Voiture.class);  
xstream.addImplicitCollection(Client.class, "voitures");
```

```
<client id="1">  
    <voiture id="1">...</voiture>  
    <voiture id="3">...</voiture>  
    <voiture id="7">...</voiture>  
</client>
```

6.7. S rialisation avec l'API Simple

6.7.1. Pr sentation

[Simple](#) est une biblioth que ressemblant   JAXB (devenue obsol te) pour la s rialisation XML d'objets Java. Elle est extr mement simple   utiliser. Il suffit d'appliquer des *annotations* sur des classes pour les rendre automatiquement s rialisables.

```
import org.simpleframework.xml.*;

@Root class Voiture {
    @Attribute    private int id;
    @Element      private String modele;
    @Element      private float prix;
}

@Root class Client {
    @Attribute    private int id;
    @ElementList  private List<Voiture> voitures;
}
```

6.7.2. Annotations

Une annotation Java, voir [wikipedia](#) est une sorte de fonction qui s'applique au code source. Le r sultat peut  tre une simple v rification comme `@Override`, ou une directive de compilation `@SuppressWarnings`, ou des propri t s de variables `@Nullable`.

Dans le cas de Simple, quelques annotations suffisent pour g n rer les instructions permettant de s rialiser la classe :

- `@Root`   placer avant le nom de la classe, pour g n rer un  l ment XML du m me nom que la classe.
- `@Attribute`   placer devant une variable membre qu'il faut  crire sous forme d'attribut.
- `@Element`,   placer devant une variable membre qu'il faut  crire sous forme d' l ment.
- `@ElementList`,   placer devant une variable membre de type collection.

6.7.3. Exemple

Voici un exemple d'annotations plac es sur une classe :

```
@Root class Voiture {
    @Attribute private int id;
    @Element   private String modele;
    @Element   private float kilometrage;
}
```

La s rialisation d'une instance produira :

```
<voiture id="321">
```

```
<modele>Porsche 911</modele>
<kilometrage>124999</kilometrage>
</voiture>
```

6.7.4. Sérialisation

La méthode toXML s'écrit alors :



```
import org.simpleframework.xml.*;
import org.simpleframework.xml.core.*;
import java.io.File;

public void toXML(String filename) throws Exception
{
    Serializer serializer = new Persister();
    File output = new File(filename);
    serializer.write(this, output);
}
```

Pour la lancer :

```
maVoiture.toXML("voiture.xml");
```

6.7.5. Désérialisation

La méthode fromXML s'écrit :



```
public static Voiture fromXML(String filename) throws Exception
{
    Serializer serializer = new Persister();
    File input = new File(filename);
    return serializer.read(Voiture.class, input);
}
```

Pour la lancer :

```
Voiture maVoiture = Voiture.fromXML("voiture.xml");
```

6.7.6. Ajustement de la sortie XML

Dans le cas des variables membres de type collections, on peut demander à Simple de ne pas produire un élément englobant. Il suffit de paramétrer l'annotation avec `inline=true` :

```
@Root class Client {
    @Attribute private int id;
    @Element private String adresse;

    @ElementList(inline=true) private List<Voiture> voitures;
}
```

Avec ces annotations, les différents éléments de la liste seront mis directement en dessous de l'élément représentant le client.

6.8. API DOM dans d'autres langages


6.8.1. Résumé

L'API W3C DOM existe pour de nombreux langages de programmation : JavaScript, PHP, Python, etc. Elle est d'emploi quasiment identique. À part les différences de syntaxe, il faut savoir que de nombreuses fonctions comme `document.getDocumentElement()`, `element.getChildNodes()` sont remplacées par des accès directs aux propriétés : `document.documentElement`, `element.childNodes` en Python et JavaScript.

Consulter par exemple les documentations de la classe Node :


- en [Java](#),
- en [JavaScript](#),
- en [Python](#).

6.8.2. Création d'un document XML en JavaScript

Pour illustrer l'API en JavaScript, voici d'abord la création d'un document XML, et pour commencer, le cadre général : 

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <script type="text/javascript">
      <!-- FONCTIONS JAVASCRIPT ICI -->
    </script>
  </head>
  <body onload="main()">
    <p>Tapez CTRL U pour voir le source...</p>
    <pre id="affichage"></pre>
  </body>
</html>
```

6.8.3. Script de création d'un document

Voici tout d'abord la création du document XML avec une racine appelée "voitures" : 

```
function main() {
  let URI = "";
  let nomracine = "voitures";
  let Doctype = null;
  let XMLdoc = document.implementation.createDocument(
    URI, nomracine, Doctype);
  let racine = XMLdoc.documentElement;
```


On peut fournir un URI pour placer tous les éléments dans un *namespace*. Il faut alors mettre le même préfixe à tous les éléments de cet URI et les créer avec `createElement(URI, "préfixe:nom")`

NB: la variable ne peut pas s'appeler `document` car c'est le nom du document HTML dans le navigateur.

6.8.4. Création d'éléments

L'ajout d'éléments, d'attributs et de textes ressemble à ce qu'on fait en Java :



```
let voiture1 = XMLdoc.createElement("voiture");
voiture1.setAttribute("marque", "Renault");
racine.appendChild(voiture1);

let voiture2 = XMLdoc.createElement("voiture");
voiture2.appendChild(XMLdoc.createTextNode("Peugeot"));
racine.appendChild(voiture2);
```

6.8.5. Affichage du résultat

Pour finir, le résultat peut être affiché dans le document HTML par un *serializer* :



```
let serializer = new XMLSerializer();
let xml = serializer.serializeToString(XMLdoc);
xml = xml.replace(/&/g, "&amp;");
xml = xml.replace(/</g, "&lt;");
xml = xml.replace(/>/g, "&gt;");
xml = xml.replace(/\"/g, "&quot;");
xml = xml.replace(/\'/g, "&apos;");
document.getElementById("affichage").innerHTML = xml;
}
```

Notez le remplacement de certains caractères par les entités HTML.

6.8.6. Parcours d'un fichier XML

On en arrive au plus utile dans un client HTTP, l'utilisation de données reçues du réseau, en général par AJAX.



```
function main() {
    var requete = new XMLHttpRequest();
    requete.onreadystatechange = function() {
        if (requete.readyState == 4 && requete.status == 200) {
            TraiterReponse(requete.responseXML);
        }
    }
    requete.open("GET", "voitures.xml", true);
    requete.send();
}
```

La demande de téléchargement et la réponse du serveur sont asynchrones. Lorsque le fichier arrive, ça appelle `TraiterReponse`.

6.8.7. Traitement de la réponse HTTP

Par exemple, on compte les éléments `<voiture>` (méthode 1) :



```
function TraiterReponse(document) {
    let racine = document.documentElement;
    let nbvoitures = 0;
    let courant = racine.firstChild;
    while (courant != null) {
        if (courant.nodeType == Node.ELEMENT_NODE &&
            courant.nodeName == "voiture") {
            nbvoitures = nbvoitures + 1;
        }
        courant = courant.nextSibling;
    }
    document.getElementById("affichage").innerHTML =
        nbvoitures+" voitures";
}
```

6.9. Validation en JAVA

6.9.1. Présentation

L'API Java `javax.xml.validation` fournit tout ce qui permet de valider un document XML contre une DTD ou un Schéma.

Pour valider par une DTD, c'est très simple, il faut qu'il mentionne sa DTD dans une balise `<!DOCTYPE>` et il suffit de l'ouvrir ainsi :



```
// créer un constructeur avec validation
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
// lire le fichier pour remplir le document
Document document = builder.parse(new File("document.xml"));
```

Toute exception indique qu'il n'est pas valide.

6.9.2. Validation par un schéma

Par exemple, pour valider `document.xml` par `document.xsd` :



```
// lire le document xml
DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = builder.parse(new File("document.xml"));
```

```
// créer un validateur basé sur le schéma
SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Source schemaFile = new StreamSource(new File("document.xsd"));
Schema schema = factory.newSchema(schemaFile);
Validator validator = schema.newValidator();
// valider le document par le schéma
try {
    validator.validate(new DOMSource(document));
} catch (SAXException e) {
    // document non valide
}
```

Semaine 7

API SAX

Le cours de cette semaine présente :

- l'analyse d'un document XML à l'aide de l'API SAX pour Java,
- l'écriture d'un document XML en langage PHP.

7.1. Simple API for XML

7.1.1. Présentation

Cette interface de programmation permet de lire et traiter un document XML sans le stocker entièrement en mémoire. C'est au contraire de DOM qui stocke la totalité du document sous forme d'un arbre de `Node`.

SAX est destiné à traiter des documents qui sont trop gros à stocker en mémoire ou dont on n'a pas besoin de parcourir le contenu de manière aléatoire. SAX ne permet qu'un seul parcours du document, dans l'ordre dans lequel il a été enregistré.

SAX signifie *Simple API for XML*, mais aurait pu être appelée *Sequential Access for XML*.

7.1.2. Principes de SAX

Avec SAX, vous devez construire un écouteur (*listener*), c'est à dire une classe possédant certaines méthodes publiques. Cet écouteur est fourni à SAX et ses méthodes sont appelées en fonction de ce qui se trouve dans le document XML. C'est de la programmation événementielle.

C'est comme avec les interfaces Swing, vous définissez un écouteur pour les clics souris. Lorsque l'utilisateur clique, cela appelle la méthode que vous avez définie.

Avec SAX, votre écouteur doit implémenter les méthodes de l'interface `org.xml.sax.ContentHandler` ou sous-classer `org.xml.sax.DefaultHandler` qui en est une implémentation par défaut.

7.1.3. Fonctionnement de SAX

SAX parcourt le document et le découpe en fragments : balises ouvrantes, balises fermantes, textes... À chaque fragment rencontré, il appelle une méthode spécifique de l'écouteur.

Voir la figure 16, page 109.

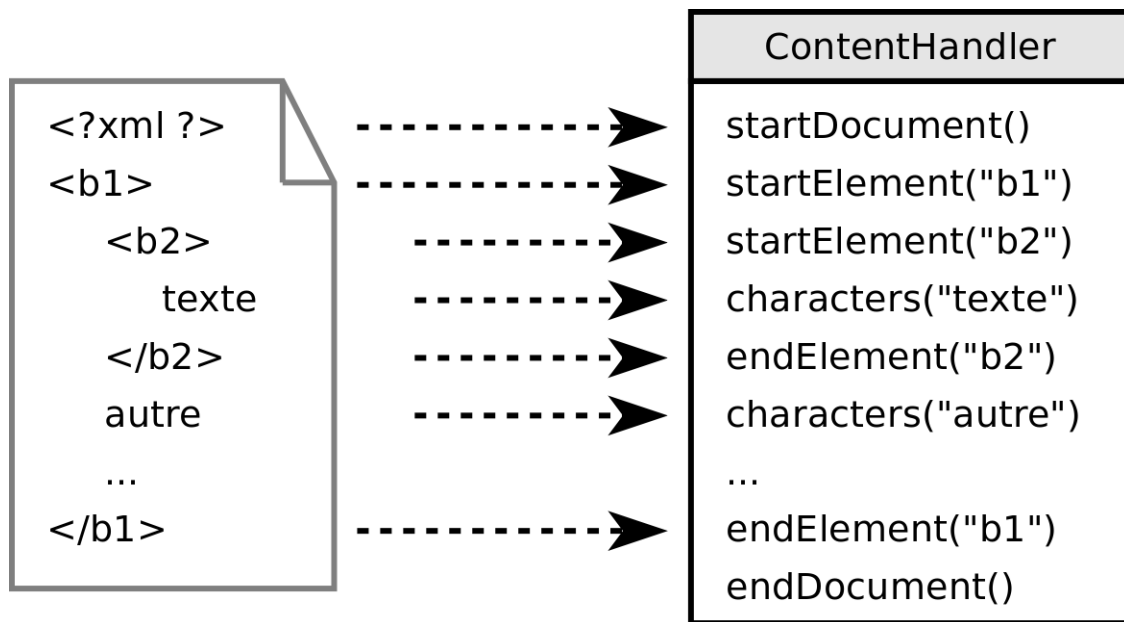


Figure 16: Principes SAX

7.1.4. Interface ContentHandler

Cette interface Java définit ce que doit implémenter un écouteur SAX. Ce sont [11 méthodes](#), dont :

- void [startDocument\(\)](#) : appelée quand on est au début du document
- void [endDocument\(\)](#) : appelée à la fin du document
- void [startElement\(String uri, String localName, String qName, Attributes attrs\)](#) : on arrive sur une balise ouvrante, le paramètre qName est son nom qualifié (préfixe:nom local), attrs contient la liste des attributs, voir le transparent suivant.
- void [endElement\(String uri, String localName, String qName\)](#) : appelée quand on arrive sur une balise fermante dont le nom qualifié est qName.

7.1.5. Type Attributes

Le type [Attributes](#) mentionné dans [startElement](#) représente un tableau d'attributs :

- String [getValue\(String nomqual\)](#) retourne la valeur de l'attribut ayant ce nom qualifié (préfixe:nom)
- String [getValue\(String uri, String nom\)](#) retourne la valeur de l'attribut ayant cet URI pour identifiant de préfixe et ce nom local.

Il y a d'autres méthodes pour parcourir les attributs un par un :

- String [getLength\(\)](#) retourne le nombre d'attributs
- String [getQName\(int i\)](#) retourne le nom qualifié du i^e attribut
- String [getValue\(int i\)](#) retourne la valeur du i^e attribut

7.1.6. Interface ContentHandler (suite)

Suite des méthodes d'un ContentHandler :

- void `characters(char[] ch, int start, int length)` : signale une zone de texte qui est à extraire du tableau `ch` par :
`String texte = new String(ch, start, length);`
 - Attention : les zones de texte sont tout ce qu'il y a entre deux balises, y compris les retours à la ligne et espaces d'indentation.
Donc il faudra nettoyer les chaînes lues des espaces avant et après : utiliser la méthode `trim()` des chaînes.
 - Attention aussi car les entités peuvent être (ou pas) concaténées avec les textes qui les entourent.
Donc il faudra concaténer tous les textes successifs.

7.1.7. Texte, CDATA et entités

Soit le document suivant :



```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE info [ <!ENTITY ent "texte4"> ]>
<info>texte1<![CDATA[texte2]]>texte3&ent;texte5</info>
```

Son analyse par SAX produit les événements suivants :

1. `startElement("", "info", "info", [])`
2. `characters("texte1")`
3. `characters("texte2")`
4. `characters("texte3")`
5. `characters("texte4texte5")`
6. `endElement("", "info", "info")`

On voit que l'entité `&ent;` a été remplacée par sa valeur et concaténée avec `texte5` mais pas avec `texte3`.

7.2. Programmation d'un analyseur

7.2.1. Implémentation d'un ContentHandler

Pour traiter la plupart des documents, on peut se contenter de définir `startElement`, `endElement` et `characters` et dériver la classe `DefaultHandler` qui implémente `ContentHandler` :



```
class MonHandler extends DefaultHandler {
    public void startElement(...) throws SAXException {
        ...
    }
    public void endElement(...) throws SAXException {
        ...
    }
    public void characters(char[] text, int debut, int lng) {
        String texte = new String(text, debut, lng);
        ...
    }
}
```

```
}  
}
```

7.2.2. Lancement de l'analyse

Ensuite, voici comment on lance le travail sur un URL :



```
void Analyser(String documentURL) throws Exception {  
    // créer un générateur d'analyseur  
    SAXParserFactory factory = SAXParserFactory.newInstance();  
    factory.setNamespaceAware(true);  
    factory.setValidating(true);  
    // créer un analyseur  
    SAXParser parser = factory.newSAXParser();  
    // créer un écouteur qui sera activé par l'analyseur  
    MonHandler handler = new MonHandler();  
    // lancer l'analyse sur l'URI : fichier ou http://...  
    parser.parse(documentURL, handler);  
}
```

documentURL est le nom d'un fichier ou un URL sur le réseau.

7.2.3. Gestion des erreurs

La classe DefaultHandler implémente l'interface `ErrorHandler` qui récupère les exceptions provoquées par les erreurs. Le problème est qu'elle n'affiche rien. Alors en général, on surcharge au moins la méthode `fatalError` :



```
class MonHandler extends DefaultHandler {  
    ...  
    public void fatalError(SAXParseException e) {  
        System.err.println(  
            "Erreur fatale ligne "+e.getLineNumber()  
            +" colonne "+e.getColumnNumber());  
        System.err.println(e.getMessage());  
    }  
}
```

Après une erreur fatale, l'analyse s'arrête définitivement.

7.3. Traitement d'un document XML

7.3.1. Aucune visibilité globale

La conséquence du fonctionnement événementiel de SAX, c'est qu'on n'a aucune vision globale du document. Par exemple dans :

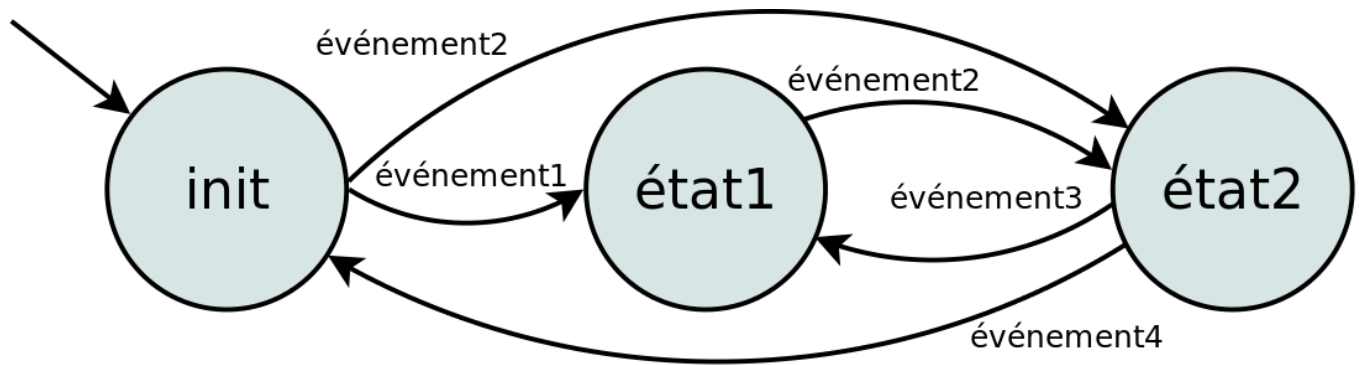


Figure 17: Automate à états

```
<voiture id="871"><prix monnaie="yen">331212</prix>...
```

Voici les événements déclenchés en séquence mais indépendamment les uns des autres :

1. `startElement("", "voiture", "voiture", [id="871"])`
2. `startElement("", "prix", "prix", [monnaie="yen"])`
3. `characters("331212")`
4. `endElement("", "prix", "prix")`

Comment faire pour convertir le prix en euros et l'associer à la voiture ? Quand on est dans la méthode `characters`, on ne dispose plus des attributs de la balise ouvrante `prix`.

7.3.2. Mémoriser les informations au passage

Le principe est de mémoriser certaines informations pendant le parcours des données :

- Il faut mémoriser les informations dont on a besoin. Par exemple, quand on rencontre l'élément `prix`, il faut mémoriser la valeur de l'attribut `monnaie` ou le taux de change, afin de pouvoir faire la conversion au moment où on rencontrera le texte de la valeur.
- Il faut aussi gérer un *état* indiquant où on se trouve dans l'arbre XML sous-jacent, par exemple, pour savoir quand on est dans le texte de l'élément `<prix>` parce que tous les textes sont gérés par la même méthode `characters`.

Pour résoudre élégamment ces problèmes, il est recommandé de faire appel à un **automate à états**, et plus particulièrement une **Machine de Mealy**.


7.3.3. Automate à états

Un **automate à états finis** est un mécanisme abstrait possédant différents états possibles, et l'un d'entre eux est l'*état courant*. La machine peut passer d'un état à l'autre, mais c'est défini par une liste de *transitions* possibles entre ses états, déclenchées par des événements. Au début la machine est dans l'un des états désigné comme étant l'état initial.

On représente une telle machine par un graphe. Les nœuds sont les états et les arcs sont les transitions possibles. Exemple :

Voir la figure 18, page 114.

7.3.4. Programmation d'un automate à états

C'est assez simple. Chaque état est représenté par un code. Il y a un écouteur par événement et un aiguillage selon l'état courant : 

```
private enum Etat {INIT, ETAT1, ETAT2};
private Etat m_EtatCourant = INIT;
public void onEvenement1() {
    switch (m_EtatCourant) {
        case INIT: m_EtatCourant = Etat.ETAT1; break;
        default: break;
    }
}
public void onEvenement2() {
    switch (m_EtatCourant) {
        case INIT: m_EtatCourant = Etat.ETAT2; break;
        case ETAT1: m_EtatCourant = Etat.ETAT2; break;
        default: break;
    }
}
```

7.3.5. Machine de Mealy

Tel quel, un automate à état ne peut pas faire grand chose. Une [Machine de Mealy](#) définit en plus des traitements à faire sur les transitions. L'arrivée d'un événement déclenche non seulement le passage d'un état à l'autre, mais aussi un traitement.

Par exemple quand on passe de l'état INIT à l'état ETAT2 à cause de l'événement 2, on peut mémoriser une information, incrémenter un compteur, afficher un message, etc.

On va utiliser ce dispositif dans le *Handler SAX* pour coller à la structure du fichier XML et mémoriser les informations nécessaires pour les traitements. Les états représentent les éléments du document XML et les transitions seront provoquées par les événements SAX.

7.3.6. Application à l'analyse SAX

Voici la machine qu'on pourrait définir pour gérer le document XML des voitures.

```
<voiture id="871"><prix monnaie="yen">331212</prix></voiture>
```

Voir la figure 18, page 114.

NB: toutes les transitions possibles ne sont pas dessinées, ex: `characters` dans les états INIT et VOITURE.

7.3.7. Traitements des transitions

Alors par exemple, voici une partie du traitement de l'événement `startElement` : 

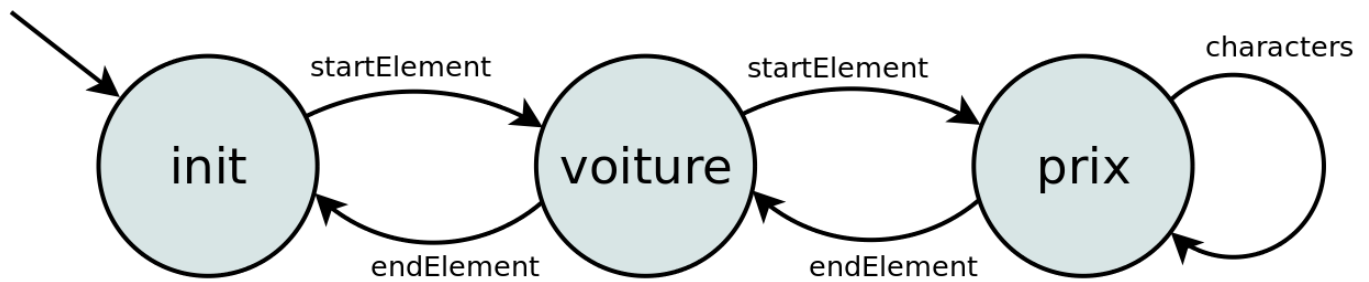



Figure 18: Machine de Mealy

```
private float m_TauxChange;
public void startElement(..., String qName, Attributes attrs) {
    switch (m_EtatCourant) {
        ...
        case VOITURE:
            if (qName.equals("prix")) {
                // calculer le taux de change s'il y a une monnaie
                String monnaie = attrs.getValue("monnaie");
                if ("yen".equals(monnaie)) m_TauxChange = 0.0082f;
                else m_TauxChange = 1.0f;
                m_EtatCourant = Etat.PRIX;
            } else throw new SAXException("balise inattendue ici");
            break;
```

7.3.8. Traitements des transitions (suite)

Et voici le traitement des événements `characters`. On se contente de mémoriser le texte dans une variable globale. Le traitement de ce texte sera fait dans l'événement `endElement` (c'est un choix personnel, on pourrait faire autrement). 

```
private String m_Texte;

public void characters(char[] text, int debut, int lng)
{
    m_Texte = new String(text, debut, lng);
}
```

- Notez que `m_EtatCourant` ne change pas : on reste dans le même état comme c'est défini dans le schéma.
- On pourrait/devrait concaténer tous les textes qui arrivent successivement, voir le transparent suivant.

7.3.9. Concaténation de tous les textes

Lorsqu'un élément contient plusieurs types de textes, comme :


```
<info>texte1<![CDATA[texte2]]>texte3&ent;texte5</info>
```

Ça va générer plusieurs événements `characters` distincts. Alors on doit concaténer les morceaux ainsi :

```
public void characters(char[] text, int debut, int lng)
{
    m_Texte.concat(new String(text, debut, lng));
}
```

Et il faut penser à réinitialiser `m_Texte` à chaque élément, dans `startElement` et dans `endElement`. Voir en TP, l'utilisation de la classe [StringBuilder](#).

7.3.10. Traitements des transitions (fin)


Pour finir, voici le traitement de l'événement `endElement`. C'est lui qui affiche le prix en € : 

```
public void endElement(..., String qName) {
    switch (m_EtatCourant) {
        case INIT:      break;
        case VOITURE:    m_EtatCourant = Etat.INIT; break;
        case PRIX:
            float prix = Float.valueOf(m_Texte) * m_TauxChange;
            System.out.println("prix = "+prix+" €");
            m_EtatCourant = Etat.VOITURE;
            break;
        default:
    }
}
```

NB: cet exemple se limite à de l'affichage, mais voir mieux en TP.

7.4. API XMLWriter de PHP

7.4.1. Présentation


L'API [XMLWriter](#) pour PHP ressemble énormément à ce qu'on vient de voir, sauf qu'elle sert à créer un document XML. Voici un court extrait pour vous convaincre : 

```
$writer = new XMLWriter();
$writer->openURI('php://output');
$writer->startDocument('1.0');
    $writer->startElement('voiture');
        $writer->startElement('prix');
            $writer->writeAttribute('monnaie', 'yen');
            $writer->text('331212');
```

```
$writer->endElement();  
$writer->endElement();  
$writer->endDocument();  
$writer->flush();
```

L'indentation permet de vérifier visuellement la structure.

7.4.2. Ouverture du flux de sortie

Le script PHP doit produire un document XML en sortie. Voici le début typique d'un tel script : 

```
<?php  
header("Content-Type: text/xml");  
$writer = new XMLWriter();  
$writer->openURI('php://output');
```

L'entête définit la nature des données émises par le script PHP. Ensuite, on crée un écrivain redirigé vers la sortie du script (c'est le même flux que echo et print).

Le script PHP se termine par :

```
$writer->flush();  
?>
```

7.4.3. Écriture d'éléments

L'API est très riche ([documentation](#)). Quelques fonctions utiles :

- `startDocument(version, encodage, standalone)` : écrit le prologue XML du document avec les paramètres optionnels fournis
- `endDocument()` : clôture le document
- `writeElement(nom, contenu)` : écrit un petit élément `<nom>contenu</nom>`
- `startElement(nom)` : écrit le début d'une balise ouvrante `<nom>`. On peut ensuite rajouter des attributs et un contenu
- `writeAttribute(nom, valeur)` : rajoute l'attribut `nom="valeur"` à l'élément actuellement ouvert
- `text(texte)` : écrit le texte, il est rajouté à l'élément courant
- `endElement()` : écrit la balise fermante `</nom>`.

Semaine 8

XML dans le SGBD PostgreSQL

Le cours de cette semaine présente la gestion et la production de données XML par un SGBD.

8.1. XML dans un SGBD

8.1.1. Présentation

Comment peut-on stocker et récupérer des données XML dans un SGBD tel que PostgreSQL ?

Ce SGBD intègre différents dispositifs permettant de gérer des données XML :

- Un type de données XML pour stocker un arbre d'éléments,
- Des fonctions pour transformer des données en XML et inversement.

Voir [la doc](#).

8.1.2. Stockage de données XML

Le type XML permet de stocker des éléments XML dans une table SQL :



```
CREATE TABLE TestXML (id INTEGER PRIMARY KEY, data XML);  
INSERT INTO TestXML VALUES (1, '<test>ok</test>');  
SELECT * FROM TestXML;
```

En fait, le type XML est quasiment identique à VARCHAR ou TEXT. Le seul avantage est la vérification du XML :



```
INSERT INTO TestXML VALUES (2, '<test>mauvais</verif>');
```

```
line 1: Opening and ending tag mismatch: test line 1 and verif  
line 1: chunk is not well balanced
```

8.1.3. Texte vers XML

Dans les exemples précédents, il y a une conversion implicite d'une chaîne en XML. Il est préférable de faire appel à la fonction XMLPARSE :



```
INSERT INTO TestXML VALUES (3,  
    XMLPARSE(DOCUMENT '<verif>oui</verif>'));  
INSERT INTO TestXML VALUES (4,  
    XMLPARSE(CONTENT 'bidule<test>fragment</test>reste'));
```

Elle prend deux paramètres :

- DOCUMENT ou CONTENT selon qu'on fournit un arbre complet ou un fragment (qui respecte la norme XML),
- une chaîne contenant le code XML à analyser.

8.1.4. Suffixe ::XML ou mot clé XML

On peut aussi suffixer les chaînes par ::XML ou les faire précéder par le mot-clé XML :

```
INSERT INTO TestXML VALUES (5, '<test>bien</test> '::XML);  
INSERT INTO TestXML VALUES (6, XML '<test>aussi</test>');
```

C'est un peu moins bien qu'appeler la fonction XMLPARSE parce qu'il n'y a pas de validation du document mais ça convient pour des contenus constants tels que ceux de l'exemple.

8.1.5. XML vers Texte

Pour le travail inverse, il y a la fonction XMLSERIALIZE :



```
SELECT id, XMLSERIALIZE(DOCUMENT data AS TEXT) FROM TestXML;  
SELECT id, XMLSERIALIZE(CONTENT data AS TEXT) FROM TestXML;
```

`XMLSERIALIZE(DOCUMENT ou CONTENT col AS TEXT|VARCHAR)`

Il faut mettre DOCUMENT si la donnée est un document XML entier (une racine et des sous-éléments), sinon il faut mettre CONTENT (plusieurs éléments).

Pour savoir si une donnée est un document ou un fragment XML :



```
SELECT id, data IS DOCUMENT FROM TestXML;
```

8.1.6. PostgreSQL et XPath

PostgreSQL permet d'employer une fonction XPath sur une colonne de type XML. Il faut utiliser la fonction PostgreSQL XPATH. Exemple :




```
SELECT XPATH(  
    '/voitures/voiture[@id=1]/couleur/text()',  
    data) FROM VoituresXML;
```

XPATH(expression, colonne)


expression expression XPath à évaluer sur le document contenu dans la colonne,
document document XML (ça ne doit pas être un fragment, mais un document complet), issu d'une colonne de table ou une chaîne constante comme ici.

retourne le résultat de l'évaluation de l'expression sur le document.

8.1.7. Génération de XML à partir de données normales

Les type et fonctions précédentes permettent de stocker du contenu XML dans une colonne de table. PostgreSQL propose [plusieurs fonctions](#) permettant de produire des documents XML à partir de colonnes ordinaires. 

```
CREATE TABLE Voitures (id INTEGER PRIMARY KEY,  
    marque TEXT, prix NUMERIC, couleur TEXT);  
INSERT INTO Voitures VALUES (1, 'Renault', 2500.00, 'blanc');  
INSERT INTO Voitures VALUES (2, 'Peugeot', 3200.00, 'gris');
```

Comment produire un document XML contenant l'extension de cette table plus facilement que comme ça : 

```
SELECT CONCAT('<voiture id=',id,'><marq>',marque,'</marq></voiture>')  
    FROM Voitures;
```

8.1.8. Génération du XML d'un n-uplet

Pour commencer, voici comment afficher un fragment XML pour chaque n-uplet de la table : 

```
SELECT XMLELEMENT(NAME "voiture", XMLATTRIBUTES(id AS "id"))  
    FROM Voitures;
```

Cela produit deux lignes, notez que ce sont des éléments vides, possédant seulement un attribut :

```
<voiture id="1"/>  
<voiture id="2"/>
```

NB: En SQL, les ' servent à délimiter des chaînes, et les " délimitent des noms de colonnes quand ces noms sont mal formés pour SQL. Ici, je mets les noms d'éléments et attributs en évidence.

8.1.9. Fonction XMLELEMENT

La fonction XMLELEMENT génère pour chaque n-uplet sélectionné un texte XML correspondant aux paramètres fournis :

XMLELEMENT(nom, attributs, contenu...)

nom il faut mettre NAME "nom" pour donner le nom de l'élément à générer

attributs ils peuvent ne pas être présents. S'il y en a, il faut employer la fonction XMLATTRIBUTES

XMLATTRIBUTES(colonne AS "nomattr", ...)

contenu il peut être absent, ou c'est une suite de XMLELEMENT et/ou de chaînes transformées en texte XML.

8.1.10. Contenu d'un XMLELEMENT

Ce qu'on met dans la partie contenu d'un XMLELEMENT peut être :

- des appels à XMLELEMENT qui seront des sous-éléments
- des chaînes de caractères qui seront transformées en texte XML
- des appels à XMLCOMMENT(texte) devenant des commentaires.

```
SELECT XMLELEMENT(NAME "parent",
    XMLELEMENT(NAME "enfant1"),
    XMLCOMMENT('blabla'),
    'texte',
    XMLELEMENT(NAME "enfant2"));
```

affiche

```
<parent><enfant1/><!--blabla-->texte<enfant2/></parent>
```

Il n'y a pas encore de fonctions pour créer des sections CDATA ainsi que des références d'entités.

8.1.11. Génération du XML d'un n-uplet (suite)

Voici par exemple la génération d'un contenu pour chaque voiture :



```
SELECT XMLELEMENT(NAME "voiture",
    XMLATTRIBUTES(id AS "id"),
    XMLELEMENT(NAME "marq", marque),
    XMLELEMENT(NAME "coul", couleur))
FROM Voitures;
```

Cela produit deux réponses, une par n-uplet :

```
<voiture id="1"><marq>Renault</marq><coul>blanc</coul></voiture>
<voiture id="2"><marq>Peugeot</marq><coul>gris</coul></voiture>
```

8.1.12. Regroupement de fragments XML

Dans les exemples précédents, on voit qu'il y a plusieurs réponses, une par n-uplet dans la base. On peut demander à agréger les réponses dans un seul arbre XML :



```
SELECT XMLELEMENT(NAME "voitures", XMLAGG(
    XMLELEMENT(NAME "voiture",
        XMLATTRIBUTES(id AS "id"),
        XMLELEMENT(NAME "marq", marque),
        XMLELEMENT(NAME "coul", couleur))))
FROM Voitures;
```

Il n'y a plus qu'une seule réponse :


```
<voitures>
  <voiture id="1"><marq>Renault</marq><coul>blanc</coul></voiture>
  <voiture id="2"><marq>Peugeot</marq><coul>gris</coul></voiture>
</voitures>
```

NB: en réalité, la réponse n'est pas indentée.

8.1.13. Regroupement de fragments XML (suite)

La fonction `XMLAGG` est une fonction d'agrégation (comme `COUNT`, `AVG`, `MAX`, `SUM`...).

`XMLAGG(contenu)`

Elle concatène toutes les réponses fournies par son paramètre pour tous les n-uplets sélectionnés. Ce paramètre doit retourner des fragments XML.

Attention `XMLAGG` ne génère pas d'élément pour englober les fragments. Donc il faut faire :

`XMLELEMENT(NAME "réponses", XMLAGG(contenu))`

8.1.14. Concaténation d'éléments

Ne pas confondre `XMLAGG` avec `XMLCONCAT`. Cette dernière concatène simplement ses paramètres. C'est implicite dans la fonction `XMLELEMENT`.

`XMLCONCAT(e1, e2, e3...)`

retourne e1 suivi de e2 suivi de e3...

```
SELECT XMLCONCAT(
  XMLELEMENT(NAME "marque", marque),
  XMLELEMENT(NAME "couleur", couleur))
FROM Voitures;
```

affiche ceci :

```
<marque>Renault</marque><couleur>blanc</couleur>
<marque>Peugeot</marque><couleur>gris</couleur>
```

8.1.15. Un contenu plus facile à écrire

Au lieu d'écrire le contenu à l'aide de plusieurs `XMLEMENT`, on peut employer `XMLFOREST` :



```
SELECT XMLEMENT(NAME "voitures", XMLAGG(  
    XMLEMENT(NAME "voiture",  
        XMLATTRIBUTES(id AS "id"),  
        XMLFOREST(marque AS "marq", couleur AS "coul"))))  
FROM Voitures;
```

`XMLFOREST(colonne1 AS "nom1", colonne2 AS "nom2", ...)`

Elle revient à écrire : `XMLCONCAT(XMLEMENT(NAME "nom1", colonne1), XMLEMENT(NAME "nom2", colonne2), ...)`

8.1.16. Entête du document

Pour finir, il manque un prologue à notre document XML. C'est le rôle de la fonction `XMLROOT` :



```
SELECT XMLROOT(  
    XMLEMENT(NAME "voitures", XMLAGG(  
        XMLEMENT(NAME "voiture",  
            XMLATTRIBUTES(id AS "id"),  
            XMLFOREST(marque AS "marq", couleur AS "coul")))),  
    VERSION '1.0')  
FROM Voitures;
```

```
<?xml version="1.0"?>  
<voitures>  
  <voiture id="1"><marq>Renault</marq><coul>blanc</coul></voiture>  
  <voiture id="2"><marq>Peugeot</marq><coul>gris</coul></voiture>  
</voitures>
```

8.1.17. Racine du document

La fonction `XMLROOT` prend trois paramètres, les deux derniers sont optionnels.

`XMLROOT(document, version, standalone)`

document ça doit être un seul élément XML, fourni par exemple par `XMLEMENT`

version mettre `VERSION '1.0'`

standalone mettre `STANDALONE YES` ou `NO` uniquement s'il y a une DTD, `YES` si elle doit être utilisée uniquement pour valider, ou `NO` si la DTD sert également à définir les valeurs par défaut et les entités. Ne pas mettre cette directive s'il n'y a pas de DTD.

8.1.18. Fournir une DTD

Actuellement PostgreSQL ne définit rien pour ajouter une DTD au document. C'est à faire à la main :



```
SELECT XMLROOT(  
    XMLCONCAT(  
        '<!DOCTYPE voitures SYSTEM "voitures.dtd">',  
        XMLELEMENT(NAME "voitures", XMLAGG(...))),  
    VERSION '1.0',  
    STANDALONE YES)  
FROM Voitures;
```

Mais en plus ça ne marche pas à ce jour. Il y a un bug qui empêche l'analyse de la balise ouvrante de la DTD. (il y a la fonction XMLPI pour générer une *processing instruction* mais rien pour une DTD).

8.2. PHP, PostgreSQL et XML

8.2.1. Présentation

On se situe sur un serveur HTTP, dans le programme PHP qui répond à une requête d'un client. Comment le script PHP peut-il envoyer des données XML ? On a deux possibilités :

- Utiliser l'API XMLWriter PHP vu au cours précédent,
- Faire encoder les données par le SGBD comme vu précédemment, PHP n'a donc presque à faire, tout est dans la requête SQL.

8.2.2. Utilisation de l'API XMLWriter

Dans ce cas, le script PHP doit :

1. Faire une requête SQL qui retourne les données
2. Utiliser un XMLWriter pour encoder les résultats

Les transparents qui suivent montrent un exemple sur la table *Voiture*.

8.2.3. Ouverture de la base

Pour commencer, on crée un objet PDO représentant la connexion avec la base de données, puis une requête SQL :

```
<?php  
$host='localhost';  
$db = 'basecontenantlatableVoitures';  
$user = 'utilisateur';  
$password = 'motdepasse';  
try {  
    $pdo = new PDO("pgsql:host=localhost;port=5432;dbname=$db",  
        $user, $password);  
  
    $sql = 'SELECT * FROM Voitures';  
    $result = $pdo->query($sql);
```

8.2.4. Création d'un écrivain XML

Ensuite, on crée un XMLWriter pour construire le document XML :

```
header("Content-Type: text/xml");
$writer = new XMLWriter();
$writer->openURI('php://output');
$writer->startDocument('1.0');
$writer->startElement('voitures');
```

On n'écrit l'entête `text/xml` que lorsqu'on est sûr qu'il va y avoir des réponses, sinon ça pourrait poser un problème avec l'affichage des messages d'erreur, voir la clause `catch`.

8.2.5. Création d'un écrivain XML

La suite consiste à écrire les n-uplets à l'aide de l'écrivain :

```
while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
    $writer->startElement('voiture');
    $writer->writeAttribute('id', $row['id']);
    $writer->writeElement('marque', $row['marque']);
    $writer->writeElement('couleur', $row['couleur']);
    $writer->endElement();
}
```

8.2.6. Terminaison

Pour finir, on ferme le document XML et on l'émet vers le client :

```
$writer->endElement();
$writer->endDocument();
$writer->flush();

} catch (PDOException $e) {
    echo $e->getMessage();
}
?>
```

8.2.7. Encodage par le SGBD

Dans cette approche, c'est le SGBD qui fait l'encodage en XML. Le script PHP doit seulement transmettre le résultat au client.

Le début est identique :

```
<?php
$host='localhost';
$db = 'basecontenatlatableVoitures';
```

```
$user = 'utilisateur';
$password = 'motdepasse';
try {
    $pdo = new PDO("pgsql:host=localhost;port=5432;dbname=$db",
        $user, $password);
```

8.2.8. Requête SQL

La requête SQL est nettement plus complexe car c'est elle qui encode en XML :



```
$sql = 'SELECT XMLROOT(
    XMLELEMENT(NAME "voitures", XMLAGG(
        XMLELEMENT(NAME "voiture",
            XMLATTRIBUTES(id AS "id"),
            XMLFOREST(marque AS "marque",
                couleur AS "couleur")))),
    VERSION \'1.0\',
    STANDALONE YES)
FROM Voitures;';
$result = $pdo->query($sql);
```

Notez les \' pour masquer les ' dans la requête.

8.2.9. Reste du script PHP

Il ne reste plus que ça, l'affichage du premier résultat de la requête (sachant qu'elle agrège les n-uplets dans un document XML complet) :



```
header("Content-Type: text/xml");
echo $result->fetch()[0];

} catch (PDOException $e) {
    echo $e->getMessage();
}
?>
```

8.2.10. Comparaisons

- XML généré par le script PHP
- XML généré par le SGBD
 - simplifie le code PHP
 - quelques bugs pour l'instant avec les entités, CDATA et DOCTYPE

8.3. Autres formats de données internet

8.3.1. Alternatives au XML

XML sert à :

- représenter des informations et permettre des recherches à l'aide de XQuery
- échanger des informations entre un serveur et un client, par exemple avec [AJAX](#)

Pour ce dernier point, il existe des alternatives :

- JSON
- YAML

8.3.2. JSON

JavaScript Object Notation est un format texte qui permet de représenter des objets complexes ([biblio](#)).

```
class Article {  
    private int id;  
    private String nom;  
    private float prix;  
    private String[] infos;  
}
```

```
{  
    "id": 1,  
    "nom": "ballon de basket",  
    "prix": 12.50,  
    "infos": ["certifié", "orange", "renforcé"]  
}
```

8.3.3. Schéma de JSON

Comme pour XML, il est possible de valider un document JSON à l'aide d'un schéma. Par exemple le document précédent répond à ce schéma, lui-même en JSON :

```
{  
    "$schema": "http://json-schema.org/draft-04/schema#",  
    "title": "Article",  
    "description": "un article de sport",  
    "type": "object",  
    "properties": {  
        "id": {  
            "description": "identifiant de l'article",  
            "type": "integer"  
        },  
    },  
}
```

8.3.4. Suite du schéma

```
    "nom": {
      "type": "string"
    },
    "prix": {
      "type": "number",
      "minimum": 0, "exclusiveMinimum": true
    },
    "infos": {
      "type": "array", "items": { "type": "string" },
      "uniqueItems": true
    }
  },
  "required": ["id", "nom", "prix"]
}
```

8.3.5. Outils de validation

Il y a des outils pour valider un document par un schéma, par exemple en ligne

<http://www.jsonschemavalidator.net/>

Pour davantage d'informations sur les schémas JSON : <http://json-schema.org/>

8.3.6. Sérialisation JSON

Il est très simple de produire un document JSON en PHP :

```
<?php
class Article {
    public $nom = "ballon de basket";
    public $prix = 12.50;
    public $infos = ["certifié", "orange", "renforcé"];
};
$article = new Article;
header('Content-type:application/json; charset=utf-8');
echo json_encode($article);
?>
```

- NB: la classe est définie d'une manière très désinvolte.
- NB: il y a un problème avec les caractères accentués.

8.3.7. Dé-sérialisation JSON

Inversement, pour récupérer un objet JavaScript à partir de JSON, il suffit de faire ceci :

```
<script>
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
```

```
        var article = JSON.parse(xmlhttp.responseText);
        ...
    }
};
xmlhttp.open("GET", "http://serveur/article.php", true);
xmlhttp.send();
```

C'est beaucoup plus simple qu'analyser du XML.

8.3.8. YAML

YAML est un format de représentation des données similaire à JSON. YAML représente toutes les données à l'aide de liste (énumérations commençant par un -) et de dictionnaires (paires nom : valeur).

Voici un exemple :

```
nom:    "ballon de basket"
prix:   12.50
infos:
  - "certifié"
  - "orange"
  - "renforcé"
```

Comme pour JSON, il y a des outils de sérialisation et de dé-sérialisation.