

XML est un **langage de balisage générique** qui permet de structurer des données afin qu'elles soient lisibles aussi bien par les humains que par des programmes de toutes sortes.

- Imposer une structure bien précise à nos documents XML.
- Les mettre en forme.
- Lire facilement les données contenues dans un document XML.
- Transformer les documents XML vers d'autres formats comme une page Internet.

● Partie 1 - Les bases du XML

1. 1. Qu'est-ce que le XML ?
2. 2. Les bons outils
3. 3. Les éléments de base
4. 4. Votre premier document XML
5. 5. TP : structuration d'un répertoire
6. 6. Quiz : Quiz 1

● Partie 2 - Créez des définitions pour vos documents XML

1. 1. Introduction aux définitions et aux DTD
2. 2. DTD : les attributs et les entités
3. 3. DTD : où les écrire ?
4. 4. TP : définition DTD d'un répertoire
5. 5. Schéma XML : introduction
6. 6. Schéma XML : les éléments simples
7. 7. Schéma XML : les types simples
8. 8. Schéma XML : les types complexes
9. 9. Schéma XML : aller plus loin
10. 10. TP : Schéma XML d'un répertoire
11. 11. Quiz : Quiz 2
12. 12. Activité : Créez un schéma XML

● Partie 3 - Traitez vos données XML

1. 1. DOM : Introduction à l'API
2. 2. DOM : Exemple d'utilisation en Java
3. 3. XPath : Introduction à l'API

4. 4. XPath : Localiser les données
5. 5. TP : des expressions XPath dans un répertoire
6. Quiz : Quiz 3
7. Activité : Récupérez des informations grâce à XPath

● Partie 4 - Transformez vos documents XML

1. 1. Introduction à XSLT
2. 2. Les templates
3. 3. Les templates : aller plus loin
4. 4. TP : des transformations XSLT d'un répertoire
5. Activité : Transformez votre document XML en un document HTML

● Partie 5 - Annexes

1. 1. Les espaces de noms
2. 2. Mettez en forme vos documents XML avec CSS

Qu'est-ce que le XML ?

Première définition

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

Un langage informatique

Regrouper en 3 grandes catégories :

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

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

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

Les **langages de description** permettent de décrire et structurer un ensemble de données selon un jeu de règles et des contraintes définies. On peut par exemple utiliser ce type de langage pour décrire l'ensemble des livres

d'une bibliothèque, ou encore la liste des chansons d'un CD, etc. Parmi les langages de description les plus connus, on peut citer le SGML, le XML ou encore le HTML.

Un langage de balisage générique

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

Le terme **générique** signifie que nous allons pouvoir créer nos propres balises.

*Le **langage XML** est un langage qui permet de décrire des données à l'aide de **balises** et de règles que l'on peut personnaliser.*

Origine et objectif du XML

Les objectifs du XML

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

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

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

En résumé

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

Les bons outils

L'éditeur de texte

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

Sinon, il est possible d'utiliser XML Copy Editor

Les éléments de base

Une balise porte un nom qui est entouré de **chevrons**.

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

Les balises par paires

Définition

Les **balises par paires** sont composées en réalité de 2 balises que l'on appelle **ouvrantes** et **fermantes**.

<balise></balise>

XML est sensible à la casse. *Toute balise ouverte doit impérativement être fermée.*

<balise>Je suis le contenu de la balise</balise>

Quelques règles

Une **balise par paires** ne peut pas contenir n'importe quoi : elle peut contenir une **valeur simple** comme par exemple une chaîne de caractères, un nombre entier, un nombre décimal, etc; également contenir une **autre balise**. On parle alors d'**arborescence**.

<balise 1>

<balise 2>10</balise2>

</balise1>

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

<balise1>

Ceci est une chaîne de caractères

<balise2>10</balise2>

7.5

</balise1>

Les balises uniques

Une **balise unique** est en réalité une balise par paires qui n'a pas de contenu.

<balise />

Les règles de nommage des balises

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

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

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

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

Les attributs

Définition

Il est possible d'ajouter à nos balises ce qu'on appelle des **attributs**. Tout comme pour les balises, c'est vous qui en choisissez le nom.

Un **attribut** peut se décrire comme une option ou une donnée cachée. Ce n'est pas l'information principale que souhaite transmettre la balise, mais il donne des renseignements supplémentaires sur son contenu.

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

<prix devise="euro">25.3</prix>

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

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

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

Quelques règles

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

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

Les commentaires

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

Votre premier document XML

Structure d'un document XML

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

Le prologue

Le **prologue** correspond à la première ligne de votre document XML. Il donne des informations de traitement. Voici à quoi notre prologue ressemble dans cette première partie du tutoriel :

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

La version

Le jeu de caractères

Un document autonome => standalone="yes".

Le corps

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

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

Vérification du document

cf XML Copy Editor

TP : structuration d'un répertoire

Utilisation concrète de structuration de données via XML.

L'énoncé

Votre répertoire doit comprendre au moins 2 personnes. Pour chaque personne, on souhaite connaître les informations suivantes :

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

```

1  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3  <repertoire>
4      <!-- John DOE -->
5      <personne sexe="masculin">
6          <nom>DOE</nom>
7          <prenom>John</prenom>
8          <adresse>
9              <numero>7</numero>
10             <voie type="impasse">impasse du chemin</voie>
11             <codePostal>75015</codePostal>
12             <ville>PARIS</ville>
13             <pays>FRANCE</pays>
14         </adresse>
15         <telephones>
16             <telephone type="fixe">01 02 03 04 05</telephone>
17             <telephone type="portable">06 07 08 09 10</telephone>
18         </telephones>
19         <emails>
20             <email type="personnel">john.doe@wanadoo.fr</email>
21             <email type="professionnel">john.doe@societe.com</email>
22         </emails>
23     </personne>
24
25     <!-- Marie POPPINS -->
26     <personne sexe="feminin">
27         <nom>POPPINS</nom>
28         <prenom>Marie</prenom>
29         <adresse>
30             <numero>28</numero>
31             <voie type="avenue">avenue de la république</voie>
32             <codePostal>13005</codePostal>
33             <ville>MARSEILLE</ville>
34             <pays>FRANCE</pays>
35         </adresse>
36         <telephones>
37             <telephone type="professionnel">04 05 06 07 08</telephone>
38         </telephones>
39         <emails>
40             <email type="professionnel">contact@poppins.fr</email>
41         </emails>
42     </personne>
43 </repertoire>
44

```

Quelques explications

Le sexe

Dans un attribut de la balise `<personne/>` et non d'en faire une balise à part entière.

En effet, cette information est, je pense, plus utile à l'ordinateur qui lira le document qu'à toute personne qui prendrait connaissance de ce fichier.

L'adresse

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

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

En effet, une machine qui traitera ce fichier, pourra facilement accéder au type de la voie sans avoir à récupérer le contenu de la balise <voie/> et tenter d'analyser s'il s'agit d'une impasse, d'une rue, d'une avenue, etc. C'est donc un gain de temps dans le traitement des données.

Numéros de téléphone et adresses e-mails

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

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

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

Introduction aux définitions et aux DTD

Le XML est très peu utilisé seul.

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

Une **définition d'un document XML** est un ensemble de règles que l'on impose au document. Ces règles permettent de décrire la façon dont le document XML doit être construit.

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

Pourquoi écrire des définitions ?

La définition impose ainsi une écriture uniforme que tout le monde doit respecter. On évite ainsi que l'écriture d'un document soit anarchique et, par conséquent, difficilement exploitable.

Exploitable le plus souvent, par un programme informatique ! Vous pouvez par exemple écrire un programme informatique qui traite les données contenues dans un document XML respectant une définition donnée.

Imposer une définition aux documents que votre programme exploite permet d'assurer un automatisme et un gain de temps précieux :

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

Deux technologies pour écrire les définitions de vos documents XML : les **DTD** ou les **schémas XML**.

Définition d'une DTD

Une **Document Type Definition** abrégé **DTD**

Où écrire les DTD ?

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

Il existe 2 types de DTD : les **DTD externes** et les **DTD internes**.

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

Les éléments

La syntaxe

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

`<!ELEMENT balise (contenu)>`

Retour sur la balise

Le mot-clef balise est à remplacer par le nom de la balise à laquelle vous souhaitez appliquer la règle.

`<nom>DOE</nom>`

=

`<!ELEMENT nom (contenu)>`

Retour sur le contenu

Cas d'une balise en contenant une autre

Par exemple, regardons la règle suivante :

`<!ELEMENT personne (nom)>`

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

Le document XML respectant cette règle ressemble donc à cela :

```
1 <personne>
2   <nom>John DOE</nom>
3 </personne>
4
```

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

Cas d'une balise contenant une valeur

Dans le cas où notre balise contient une **valeur simple**, on utilisera le mot clef `#PCDATA`

Une valeur simple désigne par exemple une chaîne de caractères, un entier, un nombre décimal, un caractère, etc.

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

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

Cas d'une balise vide

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

Cas d'une balise pouvant tout contenir

Une balise qui peut tout contenir, c'est à dire, une autre balise, une valeur simple ou tout simplement être vide.

Dans ce cas, on utilise le mot-clef `ANY`.

Structurer le contenu des balises

Nous allons voir maintenant des **syntaxes** permettant d'apporter un peu de **généricité** aux définitions DTD.

Par exemple, un répertoire contient généralement un nombre variable de personnes, il faut donc permettre au document XML d'être valide quel que soit le nombre de personnes qu'il contient.

La séquence

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

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

La liste de choix

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

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

La balise optionnelle

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

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

La balise répétée optionnelle

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

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

La balise répétée

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

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

DTD : les attributs et les entités

Les attributs

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

La syntaxe

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

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

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

Retour sur la balise et l'attribut

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

=

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

Retour sur le type

Décrire le **type** de l'attribut. Est-ce une valeur bien précise ? Est-ce du texte ? Un identifiant ?

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

Les différentes valeurs possibles pour l'attribut sont séparées par une **barre verticale** .

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

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

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

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

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

Cas d'un attribut ayant pour type un identifiant unique

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

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

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

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

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

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

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

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

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

Retour sur le mode

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

Cas d'un attribut obligatoire

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

Cas d'un attribut optionnel

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

Cas d'une valeur par défaut

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

Cas d'une constante

Enfin, il est possible **de fixer la valeur d'un attribut quand celui-ci est présent** grâce au mot clef #FIXED suivi de ladite valeur.

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

Les entités

Définition

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

Les entités générales

Définition

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

<!ENTITY nom "valeur"> on utilise la syntaxe &nom et ça donne :

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

Les entités paramètres

Définition

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

La syntaxe

<!ENTITY % nom "valeur"> = %nom et ça donne

Les entités externes

Définition

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

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

DTD : où les écrire ?

Les DTD internes

Comme je vous l'ai déjà précisé dans le premier chapitre de cette seconde partie, on distingue 2 types de DTD : Une **DTD interne** est une DTD qui est écrite dans le même fichier que le document XML. Elle est généralement spécifique au document XML dans lequel elle est écrite.

La syntaxe

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

`<!DOCTYPE racine []>`

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

Illustrons avec un exemple

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

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

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

La définition DTD est la suivante :

```
1 <!ELEMENT boutique (telephone*)>
2 <!ELEMENT telephone (marque, modele)>
3 <!ELEMENT marque (#PCDATA)>
4 <!ELEMENT modele (#PCDATA)>
5
```

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

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <!DOCTYPE boutique [
4   <!ELEMENT boutique (telephone*)>
5   <!ELEMENT telephone (marque, modele)>
6   <!ELEMENT marque (#PCDATA)>
7   <!ELEMENT modele (#PCDATA)>
8 ]>
9
10 <boutique>
11   <telephone>
12     <marque>Samsung</marque>
13     <modele>Galaxy S3</modele>
14   </telephone>
15
16   <telephone>
17     <marque>Apple</marque>
18     <modele>iPhone 4</modele>
19   </telephone>
20
21   <telephone>
22     <marque>Nokia</marque>
23     <modele>Lumia 800</modele>
24   </telephone>
25 </boutique>
26
```

Les DTD externes

Définition

Une **DTD externe** est une DTD qui est écrite dans un autre document que le document XML. Si elle est écrite dans un autre document, c'est que souvent, elle est commune à plusieurs documents XML qui l'exploitent.

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

2 types de DTD : les DTD externes **PUBLIC** et les DTD externes **SYSTEM**.

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

Les DTD externes PUBLIC

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

La syntaxe est la suivante :

<!DOCTYPE racine PUBLIC "identifiant" "url">

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
2
```

Les DTD externes SYSTEM

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

<!DOCTYPE racine SYSTEM "URI">

Le fichier XML :

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <boutique>
4   <telephone>
5     <marque>Samsung</marque>
6     <modele>Galaxy 53</modele>
7   </telephone>
8
9   <telephone>
10    <marque>Apple</marque>
11    <modele>iPhone 4</modele>
12  </telephone>
13
14  <telephone>
15    <marque>Nokia</marque>
16    <modele>Lumia 800</modele>
17  </telephone>
18 </boutique>
19
```

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

```
1 <!ELEMENT boutique (telephone*)>
2 <!ELEMENT telephone (marque, modele)>
3 <!ELEMENT marque (#PCDATA)>
4 <!ELEMENT modele (#PCDATA)>
5
```

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

```
1 <?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
2
3 <!DOCTYPE boutique SYSTEM "doc1.dtd">
4
5 <boutique>
6   <telephone>
7     <marque>Samsung</marque>
8     <modele>Galaxy 53</modele>
9   </telephone>
10
11   <telephone>
12    <marque>Apple</marque>
13    <modele>iPhone 4</modele>
14  </telephone>
15
16  <telephone>
17    <marque>Nokia</marque>
18    <modele>Lumia 800</modele>
19  </telephone>
20 </boutique>
21
```

[Retour sur le prologue](#)

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

TP : définition DTD d'un répertoire

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

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

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

Une solution


```

1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2
3  <!DOCTYPE repertoire SYSTEM "repertoire.dtd">
4
5  <repertoire>
6      <!-- John DOE -->
7      <personne sexe="masculin">
8          <nom>DOE</nom>
9          <prenom>John</prenom>
10         <adresse>
11             <numero>7</numero>
12             <voie type="impasse">impasse du chemin</voie>
13             <codePostal>75015</codePostal>
14             <ville>PARIS</ville>
15             <pays>FRANCE</pays>
16         </adresse>
17         <telephones>
18             <telephone type="fixe">01 02 03 04 05</telephone>
19             <telephone type="portable">06 07 08 09 10</telephone>
20         </telephones>
21         <emails>
22             <email type="personnel">john.doe@wanadoo.fr</email>
23             <email type="professionnel">john.doe@societe.com</email>
24         </emails>
25     </personne>
26
27     <!-- Marie POPPINS -->
28     <personne sexe="feminin">
29         <nom>POPPINS</nom>
30         <prenom>Marie</prenom>
31         <adresse>
32             <numero>28</numero>
33             <voie type="avenue">avenue de la république</voie>
34             <codePostal>13005</codePostal>
35             <ville>MARSEILLE</ville>
36             <pays>FRANCE</pays>
37         </adresse>
38         <telephones>
39             <telephone type="professionnel">04 05 06 07 08</telephone>
40         </telephones>
41         <emails>
42             <email type="professionnel">contact@poppins.fr</email>
43         </emails>
44     </personne>
45 </repertoire>
46

```

Le fichier DTD :

```

1 <!-- Racine -->
2 <!ELEMENT repertoire (personne*)>
3
4 <!-- Personne -->
5 <!ELEMENT personne (nom, prenom, adresse, telephones, emails)>
6 <!ATTLIST personne sexe (masculin | feminin) #REQUIRED>
7
8 <!-- Nom et prénom -->
9 <!ELEMENT nom (#PCDATA)>
10 <!ELEMENT prenom (#PCDATA)>
11
12 <!-- Bloc adresse -->
13 <!ELEMENT adresse (numero, voie, codePostal, ville, pays)>
14 <!ELEMENT numero (#PCDATA)>
15
16 <!ELEMENT voie (#PCDATA)>
17 <!ATTLIST voie type CDATA #REQUIRED>
18
19 <!ELEMENT codePostal (#PCDATA)>
20 <!ELEMENT ville (#PCDATA)>
21 <!ELEMENT pays (#PCDATA)>
22
23 <!-- Bloc téléphone -->
24 <!ELEMENT telephones (telephone+)>
25 <!ELEMENT telephone (#PCDATA)>
26 <!ATTLIST telephone type CDATA #REQUIRED>
27
28 <!-- Bloc email -->
29 <!ELEMENT emails (email+)>
30 <!ELEMENT email (#PCDATA)>
31 <!ATTLIST email type CDATA #REQUIRED>
32

```

Possible de créer de nouvelles règles = choix du type de la voie possible qu'entre rue, avenue, impasse, etc.

Schéma XML : introduction

Cette seconde technologie offre davantage de possibilités que les DTD

Les défauts des DTD

Un nouveau format

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

Le typage de données

Le second défaut que l'on retiendra dans ce cours est que les DTD ne permettent pas de typer des données.

Les apports des schémas XML

C'est pour pallier les défauts des DTD que les **Schémas XML** ont été créés.

Le typage des données

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

Les contraintes

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

Des définitions XML

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

Structure d'un schéma XML

L'extension du fichier

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

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

Le prologue

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

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

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

Le corps

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

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

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

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

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

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

Référencer un schéma XML

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

L'espace de noms

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

La location

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

Schéma XML décrivant un espace de noms

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

Schéma XML ne décrivant pas un espace de noms

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

On utilisera alors la syntaxe suivante :

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

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

Schéma XML : les éléments simples

Les éléments simples

Définition

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

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

Un **type simple**, c'est par exemple un chiffre, une date ou encore une chaîne de caractères.

Déclarer une balise comme un élément simple

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

Valeur par défaut et valeur inchangeable

Valeur par défaut

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

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

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

Valeur constante

Pour indiquer une **valeur constante**, c'est l'attribut **fixed** qui est utilisé au niveau de la balise `<element />` du Schéma XML.

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

Les attributs

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

Déclarer un attribut

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

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

Valeur par défaut, obligatoire et interchangeable

Valeur par défaut

Attribut **default** qui est utilisé au niveau de la balise<attribut />du Schéma XML.

Valeur constante

Attribut **fixed** qui est utilisé au niveau de la balise<attribut />du Schéma XML.

Attribut obligatoire

Pour indiquer qu'un attribut est *obligatoire*, nous devons renseigner la propriété **use** à laquelle nous affectons la valeur **required**.

Schéma XML : les types simples

Les types chaînes de caractères

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

Les types dates

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

Les types numériques

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

Les autres types

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

Schéma XML : les types complexes

Définition

Les éléments complexes

Un **élément complexe** est un élément qui contient d'autres éléments ou des attributs. Bien évidemment les éléments contenus dans un élément peuvent également contenir des éléments ou des attributs.

Déclarer un élément complexe

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

Les contenus des types complexes

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

- Les contenus **simples**.
- Les contenus "**standards**".
- Les contenus **mixtes**.

Les contenus simples

Définition

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

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

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

Les contenus "standards"

c'est le contenu d'un élément complexe qui n'est composé que d'autres éléments (simples ou complexes) ou uniquement d'attributs.

Balise contenant un ou plusieurs attributs

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

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

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

Balise contenant d'autres éléments

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

La séquence

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

Le type all

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

Le choix

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

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

Cas d'un type complexe encapsulant un type complexe

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

Les contenus mixtes

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

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

Un exemple

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

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

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

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

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

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

Schéma XML : aller plus loin

Le nombre d'occurrences

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

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

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

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

Le cas par défaut

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

Le nombre minimum d'occurrences

l'attribut minOccurs. Comme nous l'avons déjà vu plus haut, sa valeur par défaut est 1.

Dans le cas où nous souhaitons rendre optionnel un élément, il convient de lui affecter la valeur 0.

Le nombre maximum d'occurrences

l'attribut maxOccurs. Comme pour le nombre minimum d'occurrences, la valeur par défaut est 1. Une nouvelle fois, dans le cas où il est utilisé, sa valeur doit obligatoirement être supérieure à zéro.

A noter qu'il est également possible de ne pas spécifier un nombre maximal d'occurrences grâce au mot clef unbounded.

La réutilisation des éléments

Pourquoi ne pas tout écrire d'un seul bloc ?

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

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <banque>
3   <!-- 1er client de la banque -->
4   <client>
5     <!-- identité du client -->
6     <identite>
7       <nom>NORRIS</nom>
8       <prenom>Chuck</prenom>
9     </identite>
10
11     <!-- liste des comptes bancaires du client -->
12     <comptes>
13       <livretA>
14         <montant>2500</montant>
15       </livretA>
16       <courant>
17         <montant>4000</montant>
18       </courant>
19     </comptes>
20   </client>
21 </banque>
```

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

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

```

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

```

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

Diviser un Schéma XML

La syntaxe

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

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


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <!-- déclaration des éléments -->
4   <xsd:element name="montant" type="xsd:double" ></xsd:element>
5
6   <xsd:element name="identite">
7     <xsd:complexType>
8       <xsd:sequence>
9         <xsd:element name="nom" type="xsd:string" ></xsd:element>
10        <xsd:element name="prenom" type="xsd:string" ></xsd:element>
11      </xsd:sequence>
12    </xsd:complexType>
13  </xsd:element>
14
15  <xsd:element name="livretA">
16    <xsd:complexType>
17      <xsd:sequence>
18        <xsd:element ref="montant" ></xsd:element>
19      </xsd:sequence>
20    </xsd:complexType>
21  </xsd:element>
22
23  <xsd:element name="courant">
24    <xsd:complexType>
25      <xsd:sequence>
26        <xsd:element ref="montant" ></xsd:element>
27      </xsd:sequence>
28    </xsd:complexType>
29  </xsd:element>
30
31  <xsd:element name="comptes">
32    <xsd:complexType>
33      <xsd:sequence>
34        <xsd:element ref="livretA" ></xsd:element>
35        <xsd:element ref="courant" ></xsd:element>
36      </xsd:sequence>
37    </xsd:complexType>
38  </xsd:element>
39
40  <xsd:element name="client">
41    <xsd:complexType>
42      <xsd:sequence>
43        <xsd:element ref="identite" ></xsd:element>
44        <xsd:element ref="comptes" ></xsd:element>
45      </xsd:sequence>
46    </xsd:complexType>
47  </xsd:element>
48
49  <!-- Schéma XML -->
50  <xsd:element name="banque">
51    <xsd:complexType >
52      <xsd:sequence>
53        <xsd:element ref="client" maxOccurs="unbounded"></xsd:element>
54      </xsd:sequence>
55    </xsd:complexType>
56  </xsd:element>
57 </xsd:schema>

```

Créer ses propres types

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

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

La syntaxe

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

Mise à jour de notre banque et ses clients

```

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

```

L'héritage

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

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

En XML, 2 types d'héritages sont possibles :

- Par **restriction**.

- Par **extension**.

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

L'héritage par restriction

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

Définition

Une **restriction** est une notion qui peut s'appliquer aussi bien aux éléments qu'aux attributs et qui permet de déterminer plus précisément la valeur attendue via la détermination d'un certain nombre de contraintes. Par exemple, jusqu'à maintenant, nous sommes capables de dire qu'un élément ou qu'un attribut doit contenir un nombre entier strictement positif. Grâce aux **restrictions**, nous allons pouvoir pousser le concept jusqu'au bout en indiquant qu'un élément ou qu'un attribut doit contenir un nombre entier strictement positif compris entre 1 et 100.

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

L'héritage par extension

Une **extension** est une notion qui permet d'ajouter des informations à un type existant. On peut, par exemple, vouloir ajouter un élément ou un attribut.

Lorsque l'on déclare une extension sur un élément, c'est toujours le mot clef "base" qui est utilisé :

Les identifiants

Nous avons déjà vu que dans un Schéma XML, il est possible d'identifier des ressources et d'y faire référence grâce aux mots clefs **ID** et **IDREF**.

Il est cependant possible d'aller plus loin et d'être encore plus précis grâce à 2 nouveaux mots clefs : **key** et **keyref**.

La syntaxe

L'élément key

Au sein d'un Schéma XML, l'élément<key />est composé :

- D'un élément<selector />contenant une expression XPath afin d'indiquer l'élément à référencer.
- D'un ou plusieurs éléments<field />contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant.

L'élément keyref

L'élément<keyref />se construit sensiblement comme l'élément<key />. Il est donc composé :

- D'un élément<selector />contenant une expression XPath afin d'indiquer l'élément à référencer.
- D'un ou plusieurs éléments<field />contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant.

TP : Schéma XML d'un répertoire

L'énoncé

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

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

Voici le document XML que nous avons construit :

```

1  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3  <repertoire>
4      <!-- John DOE -->
5      <personne sexe="masculin">
6          <nom>DOE</nom>
7          <prenom>John</prenom>
8          <adresse>
9              <numero>7</numero>
10             <voie type="impasse">impasse du chemin</voie>
11             <codePostal>75015</codePostal>
12             <ville>PARIS</ville>
13             <pays>FRANCE</pays>
14         </adresse>
15         <telephones>
16             <telephone type="fixe">01 02 03 04 05</telephone>
17             <telephone type="portable">06 07 08 09 10</telephone>
18         </telephones>
19         <emails>
20             <email type="personnel">john.doe@wanadoo.fr</email>
21             <email type="professionnel">john.doe@societe.com</email>
22         </emails>
23     </personne>
24
25     <!-- Marie POPPINS -->
26     <personne sexe="feminin">
27         <nom>POPPINS</nom>
28         <prenom>Marie</prenom>
29         <adresse>
30             <numero>28</numero>
31             <voie type="avenue">avenue de la république</voie>
32             <codePostal>13005</codePostal>
33             <ville>MARSEILLE</ville>
34             <pays>FRANCE</pays>
35         </adresse>
36         <telephones>
37             <telephone type="bureau">04 05 06 07 08</telephone>
38         </telephones>
39         <emails>
40             <email type="professionnel">contact@poppins.fr</email>
41         </emails>
42     </personne>
43 </repertoire>
44

```

Le fichier XML avec le Schéma XML référencé :

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <repertoire xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="repertoire.xsd">
4     <!-- John DOE -->
5     <personne sexe="masculin">
6         <nom>DOE</nom>
7         <prenom>John</prenom>
8         <adresse>
9             <numero>7</numero>
10            <voie type="impasse">impasse du chemin</voie>
11            <codePostal>75015</codePostal>
12            <ville>PARIS</ville>
13            <pays>FRANCE</pays>
14        </adresse>
15        <telephones>
16            <telephone type="fixe">01 02 03 04 05</telephone>
17            <telephone type="portable">06 07 08 09 10</telephone>
18        </telephones>
19        <emails>
20            <email type="personnel">john.doe@wanadoo.fr</email>
21            <email type="professionnel">john.doe@societe.com</email>
22        </emails>
23    </personne>
24
25    <!-- Marie POPPINS -->
26    <personne sexe="feminin">
27        <nom>POPPINS</nom>
28        <prenom>Marie</prenom>
29        <adresse>
30            <numero>28</numero>
31            <voie type="avenue">avenue de la république</voie>
32            <codePostal>13005</codePostal>
33            <ville>MARSEILLE</ville>
34            <pays>FRANCE</pays>
35        </adresse>
36        <telephones>
37            <telephone type="bureau">04 05 06 07 08</telephone>
38        </telephones>
39        <emails>
40            <email type="professionnel">contact@poppins.fr</email>
41        </emails>
42    </personne>
43 </repertoire>
44

```

Le fichier XSD :


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <!-- balises isolées -->
4   <xsd:element name="nom" type="xsd:string"/>
5   <xsd:element name="prenom" type="xsd:string"/>
6
7   <!-- balises d'une adresse -->
8   <xsd:element name="numero" type="xsd:string"/>
9   <xsd:element name="voie">
10     <xsd:complexType>
11       <xsd:simpleContent>
12         <xsd:extension base="xsd:string">
13           <xsd:attribute name="type">
14             <xsd:simpleType>
15               <xsd:restriction base="xsd:string">
16                 <xsd:enumeration value="impasse"/>
17                 <xsd:enumeration value="avenue"/>
18                 <xsd:enumeration value="rue"/>
19               </xsd:restriction>
20             </xsd:simpleType>
21           </xsd:attribute>
22         </xsd:extension>
23       </xsd:simpleContent>
24     </xsd:complexType>
25   </xsd:element>
26   <xsd:element name="codePostal">
27     <xsd:simpleType>
28       <xsd:restriction base="xsd:string">
29         <xsd:pattern value="[0-9]{5}"/>
30       </xsd:restriction>
31     </xsd:simpleType>
32   </xsd:element>
33   <xsd:element name="ville" type="xsd:string"/>
34   <xsd:element name="pays" type="xsd:string"/>
35
36   <!-- balise adresse -->
37   <xsd:element name="adresse">
38     <xsd:complexType>
39       <xsd:sequence>
40         <xsd:element ref="numero"/>
41         <xsd:element ref="voie"/>
42         <xsd:element ref="codePostal"/>
43         <xsd:element ref="ville"/>
44         <xsd:element ref="pays"/>
45       </xsd:sequence>
46     </xsd:complexType>
47   </xsd:element>
48

```

```

48
49 <!-- balise telephone -->
50 <xsd:element name="telephone">
51   <xsd:complexType>
52     <xsd:simpleContent>
53       <xsd:extension base="xsd:string">
54         <xsd:attribute name="type">
55           <xsd:simpleType>
56             <xsd:restriction base="xsd:string">
57               <xsd:enumeration value="fixe"/>
58               <xsd:enumeration value="portable"/>
59               <xsd:enumeration value="bureau"/>
60             </xsd:restriction>
61           </xsd:simpleType>
62         </xsd:attribute>
63       </xsd:extension>
64     </xsd:simpleContent>
65   </xsd:complexType>
66 </xsd:element>
67
68 <!-- balise telephones -->
69 <xsd:element name="telephones">
70   <xsd:complexType>
71     <xsd:sequence>
72       <xsd:element ref="telephone" maxOccurs="unbounded"/>
73     </xsd:sequence>
74   </xsd:complexType>
75 </xsd:element>
76
77 <!-- balise email -->
78 <xsd:element name="email">
79   <xsd:complexType>
80     <xsd:simpleContent>
81       <xsd:extension base="xsd:string">
82         <xsd:attribute name="type">
83           <xsd:simpleType>
84             <xsd:restriction base="xsd:string">
85               <xsd:enumeration value="personnel"/>
86               <xsd:enumeration value="professionnel"/>
87             </xsd:restriction>
88           </xsd:simpleType>
89         </xsd:attribute>
90       </xsd:extension>
91     </xsd:simpleContent>
92   </xsd:complexType>
93 </xsd:element>
94
95 <!-- balise emails -->
96 <xsd:element name="emails">
97   <xsd:complexType>
98     <xsd:sequence>
99       <xsd:element ref="email" maxOccurs="unbounded"/>
100     </xsd:sequence>
101   </xsd:complexType>
102 </xsd:element>
103

```

```

103
104 <!-- balise personne -->
105 <xsd:element name="personne">
106     <xsd:complexType>
107         <xsd:sequence>
108             <xsd:element ref="nom"/>
109             <xsd:element ref="prenom"/>
110             <xsd:element ref="adresse"/>
111             <xsd:element ref="telephones"/>
112             <xsd:element ref="emails"/>
113         </xsd:sequence>
114
115         <!-- attribut sexe -->
116         <xsd:attribute name="sexe">
117             <xsd:simpleType>
118                 <xsd:restriction base="xsd:string">
119                     <xsd:enumeration value="masculin"/>
120                     <xsd:enumeration value="feminin"/>
121                 </xsd:restriction>
122             </xsd:simpleType>
123         </xsd:attribute>
124     </xsd:complexType>
125 </xsd:element>
126
127 <!-- Schéma XML -->
128 <xsd:element name="repertoire">
129     <xsd:complexType>
130         <xsd:sequence>
131             <xsd:element ref="personne" maxOccurs="unbounded" />
132         </xsd:sequence>
133     </xsd:complexType>
134 </xsd:element>
135 </xsd:schema>
136

```

DOM : Introduction à l'API

Qu'est-ce que L'API DOM ?

La petite histoire de DOM

DOM ou **Document Object Model**, son nom complet, est ce qu'on appelle un **parseur XML**, c'est-à-dire, une technologie grâce à laquelle il est possible de lire un document XML et d'en extraire différentes informations (éléments, attributs, commentaires, etc...) afin de les exploiter.

Aujourd'hui, la plupart des langages de programmation propose leur implémentation de DOM :

Dans les chapitres suivants, les exemples seront illustrés à l'aide du langage Java.

L'arbre XML

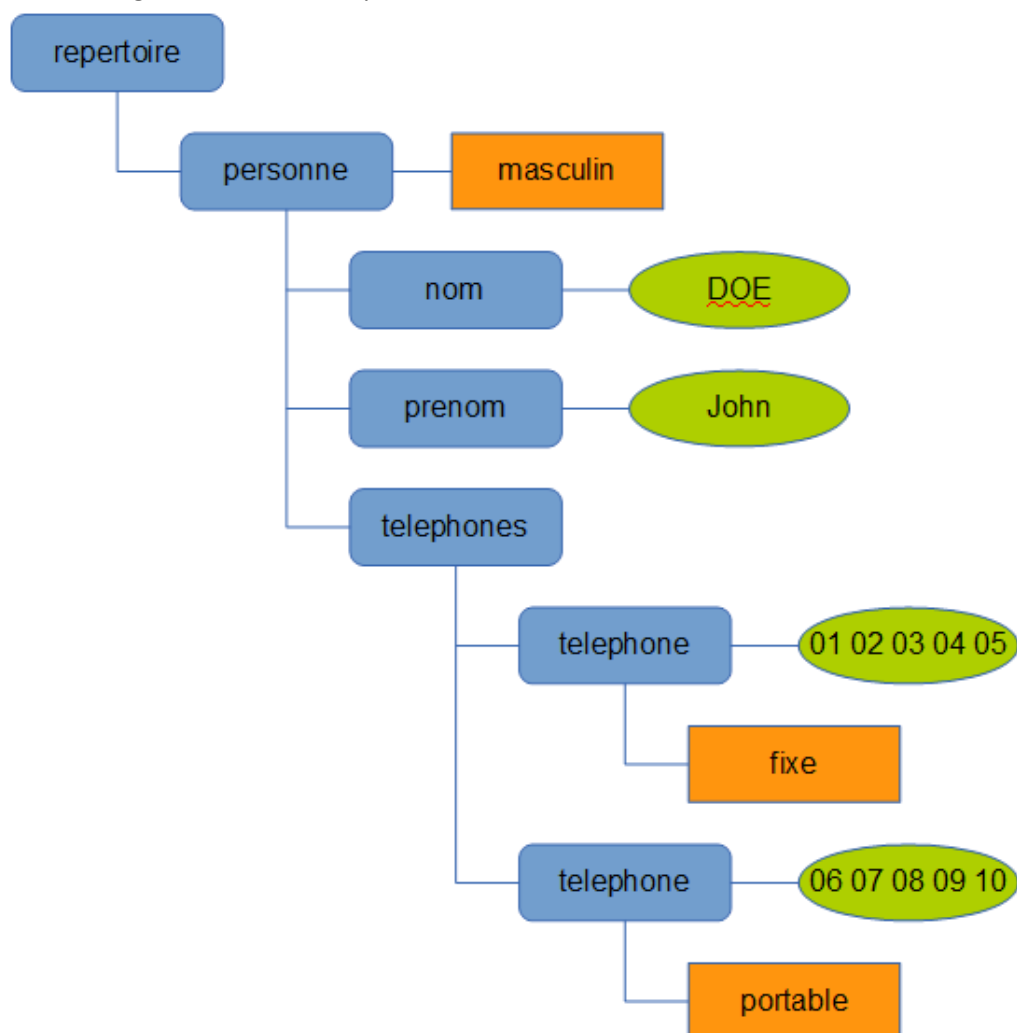
Dans le chapitre précédent, je vous disais que DOM est une technologie complètement indépendante de toute plateforme et langage de programmation et qu'elle se contente de fournir une manière d'exploiter les documents XML.

En réalité, lorsque votre document XML est lu par un parseur DOM, le document est représenté en mémoire sous la forme d'un arbre dans lequel les différents éléments sont liés les uns aux autres par une relation parent/enfant. Il est ensuite possible de passer d'un élément à un autre via un certain nombre de fonctions que nous verrons dans le chapitre suivant.

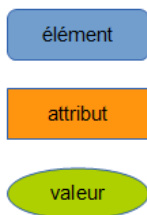
Je vous propose d'illustrer cette notion d'arbre grâce à un exemple. Soit le document XML suivant :

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</telephone>
10    </telephones>
11  </personne>
12 </repertoire>
13
```

Voici à la figure suivante ce à quoi ressemble l'arbre une fois modélisé.



Modélisation d'un arbre XML



Le vocabulaire et les principaux éléments

Document

Le **document** comme son nom le laisse deviner désigne le document XML dans son ensemble. Il est donc composé :

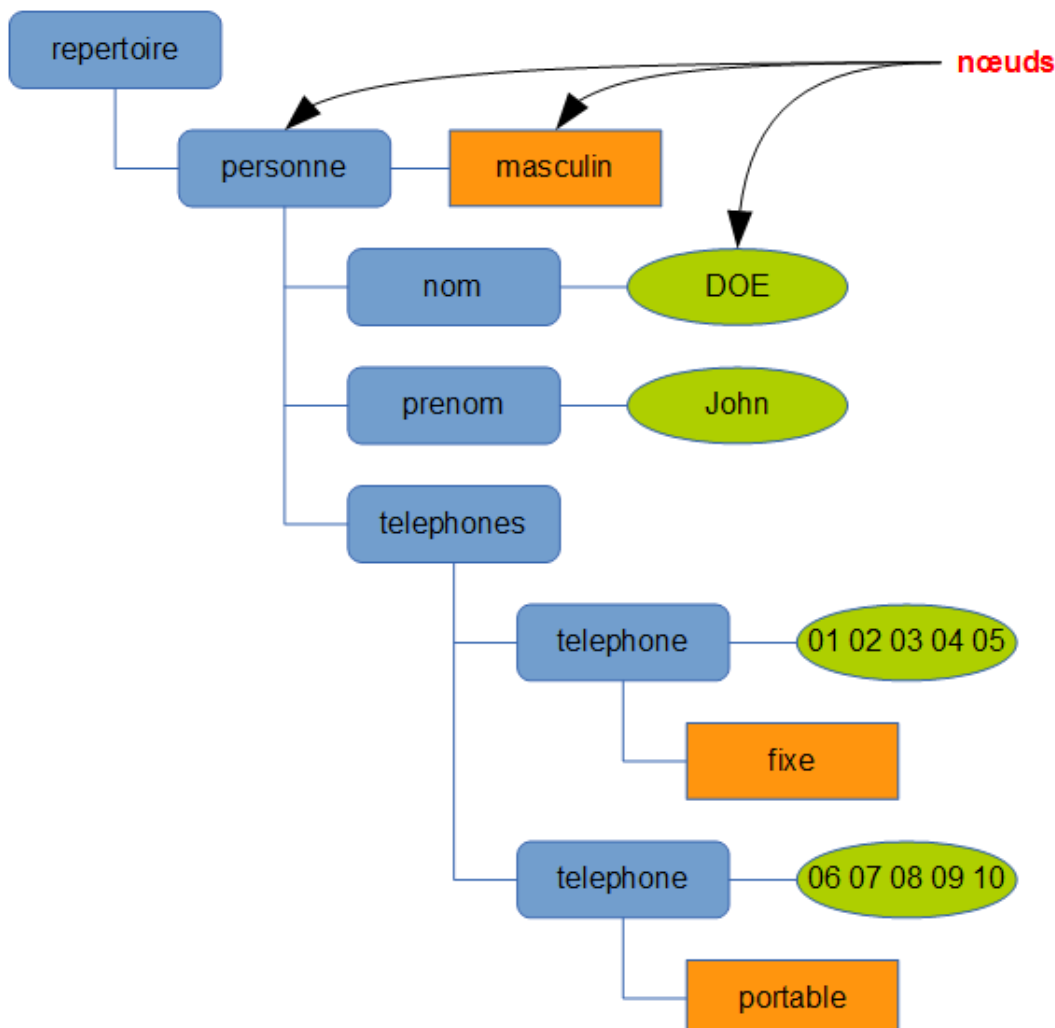
- Du **prologue**.
- Du **corps**.

La classe

Grâce à la classe **Document** nous allons pouvoir exploiter aussi bien le **prologue** que le **corps** de nos documents XML. Cette classe va également se révéler indispensable lorsque nous allons vouloir créer ou modifier des documents XML. En effet, via les nombreuses méthodes proposées, nous allons pouvoir ajouter des éléments, des commentaires, des attributs, etc.

Node

Un **Node** l'élément de base d'un arbre XML. Un élément est donc un **nœud**, tout comme une valeur et un attribut.



La classe

La classe **Node** nous permet d'obtenir un certain nombre d'informations lors de l'exploitation d'un document XML. Ainsi, il est possible d'obtenir le type du nœud (attribut, valeur, etc.) son nom, sa valeur, la liste des nœuds fils, le nœud parent, etc. Cette classe propose également un certain nombre de méthodes qui vont nous aider à créer et modifier un document XML en offrant par exemple la possibilité de supprimer un nœud, ou d'en remplacer un par un autre, etc.

Element

Définition

Un **Élément** représente une balise d'un document XML. Si l'on reprend le schéma de l'arbre XML du dessus, les éléments sont en bleu.

La classe

La classe **Element**, en plus de nous fournir le nom de la balise, nous offre de nombreuses fonctionnalités comme par exemple la possibilité de récupérer les informations d'un attribut ou encore de récupérer la listes des noeuds d'un élément portant un nom spécifique.

Attr

Définition

Un **Attr** désigne un attribut. Si l'on reprend le schéma de l'arbre XML du dessus, les attributs sont en orange.

La classe

La classe **Attr**, permet d'obtenir un certain nombre d'informations concernant les attributs comme son nom ou encore sa valeur. Cette classe va également nous être utile lorsque l'on voudra créer ou modifier des documents XML.

Text

Définition

Un **Text** désigne le contenu d'une balise. Si l'on reprend le schéma de l'arbre XML du dessus, ils sont en vert.

La classe

En plus de la donnée textuelle, la classe **Text**, permet de facilement modifier un document XML en proposant par exemple des méthodes de suppression ou de remplacement de contenu.

Les autres éléments

Il est presque impossible de présenter tous les éléments du DOM vu la densité de la technologie. Concernant les éléments non décrits, il existe par exemple la classe **Comment** permettant de gérer les commentaires ou encore la classe **CDATASection** permettant d'exploiter les sections **CDATA** d'un document XML. Finalement, sachez que grâce à DOM, il est également possible de vérifier la validité d'un document XML à une définition DTD ou un Schéma XML.

DOM : Exemple d'utilisation en Java

Lire un document XML

Le document XML

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</telephone>
10    </telephones>
11  </personne>
12 </repertoire>
13

```

Mise en place du code

Étape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"

Avant même de pouvoir prétendre créer un **parseur DOM**, nous devons récupérer une instance de la classe `DocumentBuilderFactory`. C'est à partir de cette instance que dans l'étape suivante nous pourrons créer notre parseur.

Étape 2 : création d'un parseur

La seconde étape consiste à créer un parseur à partir de notre variable `factory` créée dans l'étape 1. Une nouvelle fois, une seule ligne de code est suffisante :

Étape 3 : création d'un Document

La troisième étape consiste à créer un **Document** à partir du parseur de l'étape 2. Plusieurs possibilités s'offre alors à vous :

- A partir d'un fichier.
- A partir d'un flux.

Comme je vous le disais dans le chapitre précédent, un **Document** représente le document XML dans son intégralité. Il contient son **prologue** et son **corps**. Nous allons pouvoir le vérifier en affichant les éléments du prologue, à savoir :

- La version XML utilisée.
- L'encodage utilisé.
- S'il s'agit d'un document "standalone" ou non.

Étape 4 : récupération de l'Element racine

Dans cette quatrième étape, nous allons laisser de côté le prologue et tenter de nous attaquer au **corps** du document XML. Pour ce faire, nous allons extraire l'**Element racine** du Document. C'est à partir de lui que nous pourrons ensuite naviguer dans le reste du document.

Pour récupérer l'élément racine, il suffit de faire appel à la fonction `getDocumentElement()` de notre document :

Étape 5 : récupération des personnes

Nous allons réellement parcourir le corps de notre document XML.

Il est possible de parcourir un document XML sans connaître sa structure. Il est donc possible de créer un code assez générique notamment grâce à la récursivité. Nous allons donc parcourir notre document en partant du principe que nous connaissons sa structure.

Pour récupérer tous les noeuds enfants de la racine, voici la ligne de code à écrire :

```
final NodeList racineNoeuds = racine.getChildNodes();
```

Il vous faudra également importer le package suivant :

```
import org.w3c.dom.NodeList;
```

Nous pouvons également nous amuser à afficher le nom de chacun des nœuds via le code suivant :

```
1 final int nbRacineNoeuds = racineNoeuds.getLength();
2
3 for (int i = 0; i < nbRacineNoeuds; i++) {
4     System.out.println(racineNoeuds.item(i).getNodeName());
5 }
6
```

Vous devriez alors avoir le résultat suivant :

#text

#comment

#text

personne

#text

=> modifier notre boucle afin de n'afficher à l'écran que les nœuds étant des éléments. Grâce à la méthode `getNodeName()` de la classe **Node** :

```
1 for (int i = 0; i < nbRacineNoeuds; i++) {
2     if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE) {
3         final Node personne = racineNoeuds.item(i);
4         System.out.println(personne.getNodeName());
5     }
6 }
7
```

A l'exécution de votre programme, vous devriez normalement avoir le résultat suivant :

personne

Afficher le sexe de la personne, qui pour rappel est un attribut. Faire appel à la méthode `getAttributes()` de la classe **Node** qui nous renvoie l'ensemble des attributs du nœud.

Dans notre cas, nous allons utiliser la méthode `getAttribute(nom)` qui nous renvoie la valeur de l'attribut spécifié en paramètre. Cette méthode n'est cependant pas accessible à partir de la classe **Node**, il convient donc de "caster" notre nœud en un **Element** pour pouvoir l'appeler :

```
1 for (int i = 0; i < nbRacineNoeuds; i++) {
2     if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE) {
3         final Element personne = (Element) racineNoeuds.item(i);
4         System.out.println(personne.getNodeName());
5         System.out.println("sexe : " + personne.getAttribute("sexe"));
6     }
7 }
8
```

A l'écran devrait alors s'afficher le résultat suivant :

personne

sexe : masculin

Étape 6 : récupération du nom et du prénom

Nous allons maintenant récupérer le nom et le prénom des personnes présentes dans notre document XML. Puisque nous savons exactement ce que l'on souhaite récupérer, nous allons y accéder directement via la méthode `getElementsByTagName(name)` de l'objet **Element**. Cette méthode nous renvoie tous les éléments contenus dans l'élément et portant le nom spécifié.

Ainsi, nous venons de récupérer tous les éléments d'une personne ayant pour nom "nom". Dans notre cas, nous savons qu'une personne ne peut avoir qu'un seul nom, nous pouvons donc préciser que nous voulons le premier élément de la liste :

```
final Element nom = (Element) personne.getElementsByTagName("nom").item(0);
```

Étape 7 : récupération des numéros de téléphone

La septième et dernière étape de la lecture de notre document XML consiste à récupérer les numéros de téléphone d'une personne. La logique est sensiblement la même que dans l'étape 6 si ce n'est que le résultat de la méthode `getElementsByTagName(name)` nous renverra éventuellement plusieurs résultats. Il suffit alors de boucler sur les résultats pour afficher les valeurs et les attributs.

Le code complet

Nous venons donc de lire ensemble notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire juste en dessous :

```

1 import java.io.File;
2 import java.io.IOException;
3
4 import javax.xml.parsers.DocumentBuilder;
5 import javax.xml.parsers.DocumentBuilderFactory;
6 import javax.xml.parsers.ParserConfigurationException;
7
8 import org.w3c.dom.Document;
9 import org.w3c.dom.Element;
10 import org.w3c.dom.Node;
11 import org.w3c.dom.NodeList;
12 import org.xml.sax.SAXException;
13
14 public class ReadXMLFile {
15     public static void main(final String[] args) {
16         /*
17          * Etape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"
18          */
19         final DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
20
21         try {
22             /*
23              * Etape 2 : création d'un parseur
24              */
25             final DocumentBuilder builder = factory.newDocumentBuilder();
26
27             /*
28              * Etape 3 : création d'un Document
29              */
30             final Document document = builder.parse(new File("repertoire.xml"));
31
32             //Affichage du prologue
33             System.out.println("*****PROLOGUE*****");
34             System.out.println("version : " + document.getXmlVersion());
35             System.out.println("encodage : " + document.getXmlEncoding());
36             System.out.println("standalone : " + document.getXmlStandalone());
37
38             /*
39              * Etape 4 : récupération de l'Element racine
40              */
41             final Element racine = document.getDocumentElement();
42
43             //Affichage de l'élément racine
44             System.out.println("\n*****RACINE*****");
45             System.out.println(racine.getNodeName());
46
47             /*
48              * Etape 5 : récupération des personnes
49              */

```



```

46
47     /*
48     * Etape 5 : récupération des personnes
49     */
50     final NodeList racineNoeuds = racine.getChildNodes();
51     final int nbRacineNoeuds = racineNoeuds.getLength();
52
53     for (int i = 0; i < nbRacineNoeuds; i++) {
54         if (racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE) {
55             final Element personne = (Element) racineNoeuds.item(i);
56
57             //Affichage d'une personne
58             System.out.println("\n*****PERSONNE*****");
59             System.out.println("sexe : " + personne.getAttribute("sexe"));
60
61             /*
62             * Etape 6 : récupération du nom et du prénom
63             */
64             final Element nom = (Element) personne.getElementsByTagName("nom").item(0);
65             final Element prenom = (Element) personne.getElementsByTagName("prenom").item(0);
66
67             //Affichage du nom et du prénom
68             System.out.println("nom : " + nom.getTextContent());
69             System.out.println("prénom : " + prenom.getTextContent());
70
71             /*
72             * Etape 7 : récupération des numéros de téléphone
73             */
74             final NodeList telephones = personne.getElementsByTagName("telephone");
75             final int nbTelephonesElements = telephones.getLength();
76
77             for (int j = 0; j < nbTelephonesElements; j++) {
78                 final Element telephone = (Element) telephones.item(j);
79
80                 //Affichage du téléphone
81                 System.out.println(telephone.getAttribute("type") + " : " +
telephone.getTextContent());
82             }
83         }
84     }
85
86     catch (final ParserConfigurationException e) {
87         e.printStackTrace();
88     }
89     catch (final SAXException e) {
90         e.printStackTrace();
91     }
92     catch (final IOException e) {
93         e.printStackTrace();
94     }
95 }
96 }
97

```

Ecrire un document XML

Le document XML

Dans le chapitre précédent, nous avons vu comment lire un document XML. Dans ce chapitre, je vous propose d'en créer un de toute pièce.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</telephone>
10    </telephones>
11  </personne>
12 </repertoire>
13
```

Mise en place du code

Étape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"

Comme pour la lecture d'un document XML, la première étape consiste à récupérer une instance de la classe DocumentBuilderFactory

Étape 2 : création d'un parseur

La seconde étape est également commune à la lecture d'un document XML. Ainsi, nous allons créer un parseur à partir de notre variable factory créée dans l'étape précédente :

Étape 3 : création d'un Document

Ce document est créé à partir de notre parseur :

Étape 4 : création de l'Élément racine

Dans cette quatrième étape, nous allons créer l'élément racine de notre document XML, à savoir la balise<repertoire />. La création de l'élément racine se fait via notre document et plus particulièrement la fonctioncreateElement()qui prend en paramètre le nom que l'on souhaite donner à la balise :

Étape 5 : création d'une personne

Si l'on regarde le document XML que l'on doit créer, on s'aperçoit qu'avant de créer la balise<personne/>, nous devons créer un commentaire.

La création d'un commentaire n'est pas plus compliquée que la création d'une balise et se fait via la fonction createComment() du document qui prend comme paramètre le fameux commentaire :

```
final Comment commentaire = document.createComment("John DOE");
```

Il convient ensuite d'ajouter notre commentaire à la suite de notre document et plus spécifiquement à la suite de notre élément racine. Si l'on se réfère à l'arbre XML, le commentaire est réellement un sous élément de l'élément racine. C'est pourquoi celui-ci est ajouté en tant qu'enfant de l'élément racine :

```
racine.appendChild(commentaire);
```

Nous pouvons maintenant nous attaquer à la création de la balise<personne />. Il s'agit d'un élément au même titre que l'élément racine que nous avons déjà vu. Puisque la balise est au même niveau que le commentaire, il convient de l'ajouter en tant qu'enfant de l'élément racine :

```
final Element personne = document.createElement("personne");
racine.appendChild(personne);
```

Si l'on s'arrête ici, on omet d'ajouter l'attribut "sexe" pourtant présent dans la balise<personne />du document XML que l'on souhaite créer. Ajouter un attribut à un élément est en réalité très simple et se fait via la méthode `setAttribute()` de la classe `Element`. Cette méthode prend 2 paramètres : le nom de l'attribut et sa valeur.

```
personne.setAttribute("sexe", "masculin");
```

Étape 6 : création du nom et du prénom

En soit, la création des balises<nom />et<prenom />n'a rien de compliqué. En effet, nous avons déjà créé ensemble plusieurs éléments.

```
final Element nom = document.createElement("nom");
final Element prenom = document.createElement("prenom");
```

```
personne.appendChild(nom);
personne.appendChild(prenom);
```

Ici, la nouveauté concerne le renseignement de la valeur contenue dans les balises, à savoir John DOE dans notre exemple. Pour ce faire, il convient d'ajouter à nos balises un enfant de type `Text`. Cet enfant doit être créé avec la méthode `createTextNode()` du document qui prend en paramètre la valeur :

```
nom.appendChild(document.createTextNode("DOE"));
prenom.appendChild(document.createTextNode("John"));
```

Étape 7 : création des numéros de téléphone

Rien de nouveau par rapport à ce que nous avons vu jusqu'ici :

Étape 8 : affichage du résultat

Il est maintenant temps de passer à la dernière étape qui consiste à afficher notre document XML fraîchement créé. Deux possibilités s'offrent à nous :

- Dans un document XML.
- Dans la console de l'IDE.

Pour pouvoir afficher notre document XML, nous allons avoir besoin de plusieurs objets Java. Le premier est une instance de la classe `TransformerFactory` :

```
final TransformerFactory transformerFactory = TransformerFactory.newInstance();
```

La récupération de cette instance s'accompagne de l'importation du package suivant :

```
import javax.xml.transform.TransformerFactory;
```

Nous allons utiliser cette instance pour créer un objet `Transformer`. C'est grâce à lui que nous pourrions afficher notre document XML par la suite :

```
final Transformer transformer = transformerFactory.newTransformer();
```

Pour afficher le document XML, nous utiliserons la méthode `transform()` de notre `transformer`. Cette méthode prend en compte 2 paramètres :

- La source.
- La sortie.

```
transformer.transform(source, sortie);
```

A noter qu'une exception de type `TransformerException` est susceptible d'être levée :

```
import javax.xml.transform.TransformerException;
```

En ce qui nous concerne, la source que l'on souhaite afficher est notre document XML. Cependant, nous ne pouvons pas passer notre objet document tel quel. Il convient de le transformer légèrement sous la forme d'un objet DOMSource :

```
final DOMSource source = new DOMSource(document);
```

Pour pouvoir utiliser cette classe, il convient d'importer le package suivant :

```
import javax.xml.transform.dom.DOMSource;
```

Maintenant que nous avons la source, occupons nous de la sortie. La sortie est en réalité un objet StreamResult. C'est ici que nous allons préciser si nous souhaitons afficher notre document dans un fichier ou dans la console de notre IDE :

```
//Code à utiliser pour afficher dans un fichier
```

```
final StreamResult sortie = new StreamResult(new File("F:\\file.xml"));
```

```
//Code à utiliser pour afficher dans la console
```

```
final StreamResult sortie = new StreamResult(System.out);
```

Encore une fois, l'importation d'un package est nécessaire :

```
import javax.xml.transform.stream.StreamResult;
```

Avant d'exécuter notre programme, il nous reste encore quelques petits détails à régler : l'écriture du prologue et le formatage de l'affichage.

Commençons par le prologue. Nous allons renseigner ses différentes propriétés via la méthode `setOutputProperty()` de notre transformateur qui prend en paramètre le nom du paramètre et sa valeur :

```
transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
```

```
transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
```

```
transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
```

Pour pouvoir utiliser les constantes de la classe `OutputKeys`, il convient d'importer le package suivant :

```
import javax.xml.transform.OutputKeys;
```

Si nous exécutons notre programme maintenant, tout sera écrit sur une seule ligne, ce qui n'est pas très lisible, vous en conviendrez. C'est pourquoi nous allons donner quelques règles de formatage à notre transformateur. En effet, ce que l'on souhaite c'est que notre document soit indenté. Chaque niveau différent de notre document XML sera alors décalé de 2 espaces :

```
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
```

```
transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
```

Le code complet

Nous venons donc de créer notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire juste en dessous :

```

1  import java.io.File;
2
3  import javax.xml.parsers.DocumentBuilder;
4  import javax.xml.parsers.DocumentBuilderFactory;
5  import javax.xml.parsers.ParserConfigurationException;
6  import javax.xml.transform.OutputKeys;
7  import javax.xml.transform.Transformer;
8  import javax.xml.transform.TransformerConfigurationException;
9  import javax.xml.transform.TransformerException;
10 import javax.xml.transform.TransformerFactory;
11 import javax.xml.transform.dom.DOMSource;
12 import javax.xml.transform.stream.StreamResult;
13
14 import org.w3c.dom.Comment;
15 import org.w3c.dom.Document;
16 import org.w3c.dom.Element;
17
18
19 public class ReadXMLFile {
20     public static void main(final String[] args) {
21         /*
22          * Etape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"
23          */
24         final DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
25
26         try {
27             /*
28              * Etape 2 : création d'un parseur
29              */
30             final DocumentBuilder builder = factory.newDocumentBuilder();
31
32             /*
33              * Etape 3 : création d'un Document
34              */
35             final Document document = builder.newDocument();
36
37             /*
38              * Etape 4 : création de l'Element racine
39              */
40             final Element racine = document.createElement("repertoire");
41             document.appendChild(racine);
42
43             /*
44              * Etape 5 : création d'une personne
45              */
46             final Comment commentaire = document.createComment("John DOE");
47             racine.appendChild(commentaire);
48
49             final Element personne = document.createElement("personne");
50             personne.setAttribute("sexe", "masculin");
51             racine.appendChild(personne);
52
53             /*
54              * Etape 6 : création du nom et du prénom
55              */
56             final Element nom = document.createElement("nom");
57             nom.appendChild(document.createTextNode("DOE"));
58
59             final Element prenom = document.createElement("prenom");
60             prenom.appendChild(document.createTextNode("John"));
61
62             personne.appendChild(nom);
63             personne.appendChild(prenom);
64
65             /*
66              * Etape 7 : création des numéros de téléphone
67              */

```

```

66      * Etape 7 : création des numéros de téléphone
67      */
68      final Element telephones = document.createElement("telephones");
69
70      final Element fixe = document.createElement("telephone");
71      fixe.appendChild(document.createTextNode("01 02 03 04 05"));
72      fixe.setAttribute("type", "fixe");
73
74      final Element portable = document.createElement("telephone");
75      portable.appendChild(document.createTextNode("06 07 08 09 10"));
76      portable.setAttribute("type", "portable");
77
78      telephones.appendChild(fixe);
79      telephones.appendChild(portable);
80      personne.appendChild(telephones);
81
82      /*
83      * Etape 8 : affichage
84      */
85      final TransformerFactory transformerFactory = TransformerFactory.newInstance();
86      final Transformer transformer = transformerFactory.newTransformer();
87      final DOMSource source = new DOMSource(document);
88      final StreamResult sortie = new StreamResult(new File("F:\\\\file.xml"));
89      //final StreamResult result = new StreamResult(System.out);
90
91      //prologue
92      transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
93      transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
94      transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
95
96      //formatage
97      transformer.setOutputProperty(OutputKeys.INDENT, "yes");
98      transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
99
100     //sortie
101     transformer.transform(source, sortie);
102 }
103 catch (final ParserConfigurationException e) {
104     e.printStackTrace();
105 }
106 catch (TransformerConfigurationException e) {
107     e.printStackTrace();
108 }
109 catch (TransformerException e) {
110     e.printStackTrace();
111 }
112 }
113 }
114

```

Vous savez maintenant lire et écrire un document XML grâce à l'implémentation Java de l'API DOM.

XPath : Introduction à l'API

Qu'est-ce que l'API XPath ?

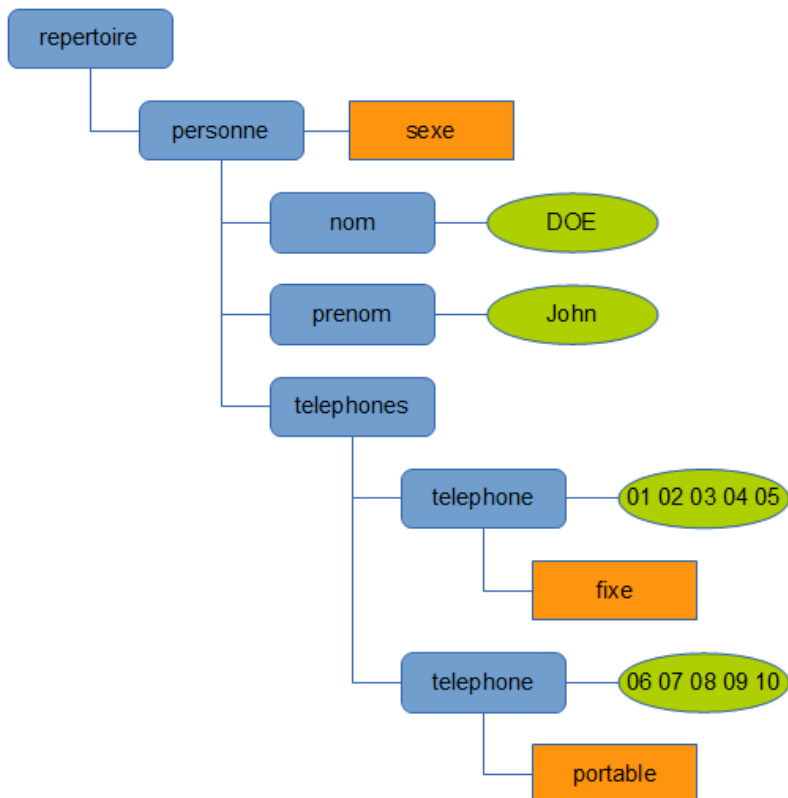
XPath est une technologie qui permet d'extraire des informations (éléments, attributs, commentaires, etc...) d'un document XML via l'écriture d'expressions dont la syntaxe rappelle les expressions rationnelles utilisées dans d'autres langages.

Tout comme DOM, XPath est un standard du W3C et ce depuis sa première version en 1999. Si XPath n'est pas un langage de programmation en soit, cette technologie fournit tout un vocabulaire pour écrire des expressions permettant d'accéder directement aux informations souhaitées sans avoir à parcourir tout l'arbre XML.

Un peu de vocabulaire

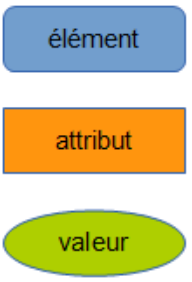
```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <telephones>
8       <telephone type="fixe">01 02 03 04 05</telephone>
9       <telephone type="portable">06 07 08 09 10</telephone>
10    </telephones>
11  </personne>
12 </repertoire>
13
```

Reprenons également une illustration de son arbre :



Arbre XML d'un document XML

Pour rappel, voici la légende :



élément

attribut

valeur

Légende de l'arbre XML

Parent

Le **parent** d'un nœud est le nœud qui est directement au dessus de lui d'un point de vue hiérarchique. Chaque nœud a au moins un parent.

Par exemple, le nœud **repertoire** est le parent du nœud **personne** qui est lui même le parent des noeuds **nom**, **prenom** et **telephones**.

Enfant

Un nœud a pour **enfants** tous les noeuds situés un niveau en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité d'enfants.

Par exemple, le nœud **repertoire** a pour enfant le nœud **personne** qui a lui même plusieurs enfants : les noeuds **nom**, **prenom** et **telephones**.

Descendant

Un nœud a pour **descendants** tous les noeuds situés en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité de descendants.

Par exemple, le nœud **repertoire** a pour descendants les nœuds **personne**, **nom**, **prenom** et **telephones**.

Ancêtre

Un nœud a pour **ancêtres** tous les noeuds situés en dessus dans la hiérarchie. Un nœud peut donc avoir plusieurs ancêtres.

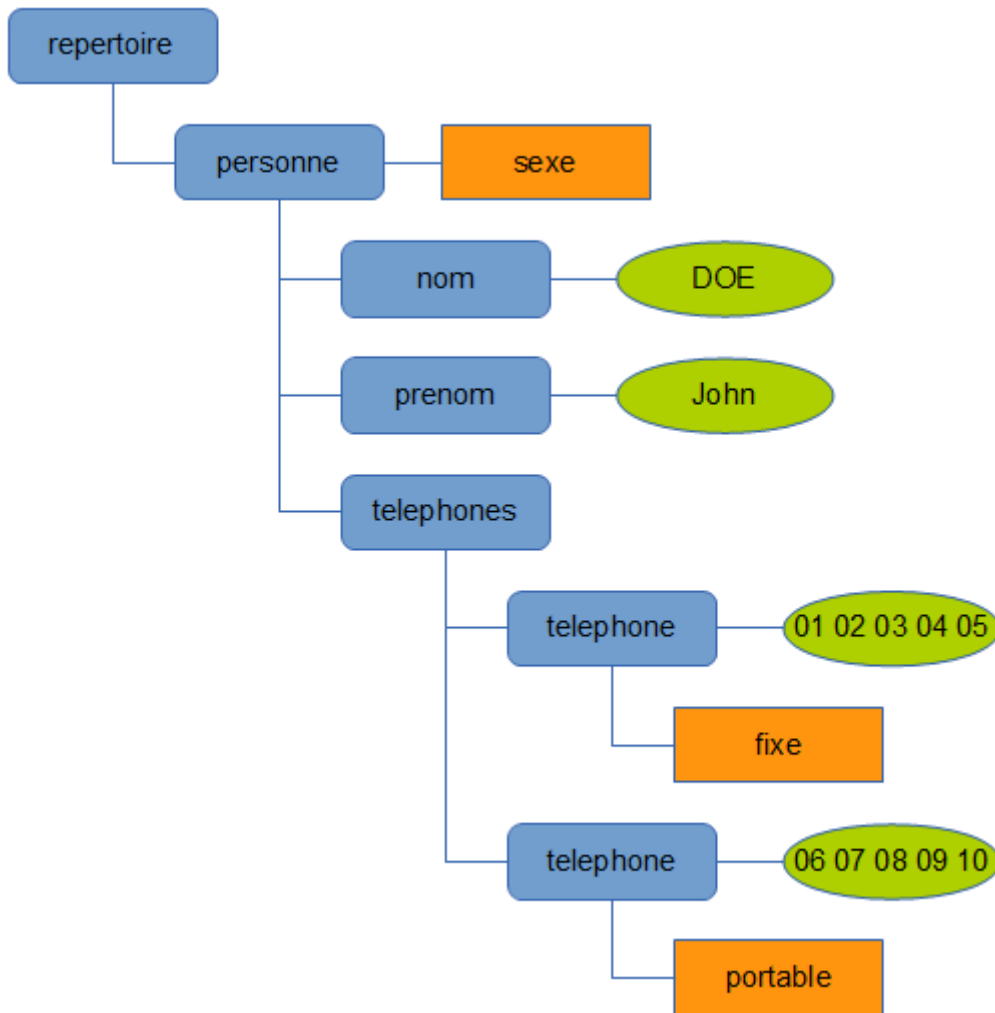
Par exemple, le nœud **telephones** a pour ancêtres les nœuds **personne** et **repertoire**.

Frère

Un nœud a pour **frères** tous les noeuds situés au même niveau dans la hiérarchie. Un nœud peut donc avoir une infinité de frères.

Par exemple, le nœud **nom** a pour frères les nœuds **prenom** et **telephones**.

Chemin relatif et chemin absolu



Arbre XML d'un document XML

Si je veux récupérer par exemple le numéro de téléphone fixe, voici le chemin à parcourir :

- Etape 1 : nœud "**repertoire**".
- Etape 2 : descendre au nœud enfant "**personne**".
- Etape 3 : descendre au nœud enfant "**telephones**".
- Etape 4 : descendre au nœud enfant "**telephone**" dont l'attribut est "**fixe**".

Sans rentrer dans les détails, l'expression XPath correspondante ressemblera à quelque chose comme ça :
/étape1/étape2/étape3/étape4

Les chemins absolus

Le nœud de départ est toujours la racine de l'arbre XML.

Une expression XPath utilisant un chemin absolu est facilement identifiable car elle commence par le caractère "/".

Les chemins relatifs

Un **chemin relatif** accepte quant à lui n'importe quel nœud de l'arbre XML comme point de départ.

Une expression XPath utilisant un chemin relatif est facilement identifiable car elle ne commence pas par le caractère "/".

XPath : Localiser les données

Dissection d'une étape

Dans le chapitre précédent, nous avons vu qu'une expression XPath est en réalité une succession d'étapes. Nous allons maintenant nous intéresser de plus près à ce qu'est une **étape**.

Une étape est décrite par 3 éléments :

- Un **axe**.
- Un **nœud** ou un type de nœud.
- Un ou plusieurs **prédicats** (facultatif).

L'**axe** va nous permettre de définir le *sens* de la recherche. Par exemple, si l'on souhaite se diriger vers un nœud enfant ou au contraire remonter vers un nœud parent voir un ancêtre.

Le **nœud** va nous permettre d'affiner notre recherche en indiquant explicitement le **nom d'un nœud** ou le **type de nœud** dont les informations nous intéressent.

Les prédicats

Facultatif. Les **prédicats**, dont le nombre n'est pas limité, agissent comme un filtre et vont nous permettre de gagner en précision lors de nos recherches. Ainsi, grâce aux **prédicats**, il sera par exemple possible de sélectionner les informations à une position précise.

Les axes

Comme nous l'avons vu dans la partie précédente, un **axe** est le premier élément formant une étape. Son rôle est de définir le *sens* de la recherche. Bien évidemment, le choix du sens est structuré par un vocabulaire précis que nous allons étudier maintenant.

Nom de l'axe	Description
ancestor	oriente la recherche vers les ancêtres du nœud courant
ancestor-or-self	oriente la recherche vers le nœud courant et ses ancêtres
attribute	oriente la recherche vers les attributs du nœud courant
child	oriente la recherche vers les enfants du nœud courant
descendant	oriente la recherche vers les descendants du nœud courant
descendant-or-self	oriente la recherche vers le nœud courant et ses descendants
following	oriente la recherche vers les nœuds suivant le nœud courant
following-sibling	oriente la recherche vers les frères suivants du nœud courant
parent	oriente la recherche vers le père du nœud courant
preceding	oriente la recherche vers les nœuds précédant le nœud courant
preceding-sibling	oriente la recherche vers les frères précédents du nœud courant
self	oriente la recherche vers le nœud courant

Il existe également un axe nommé **namespace** qui permet d'orienter la recherche vers un espace de noms. L'axe **child** Il s'agit de l'axe par défaut.

Les tests de nœuds

il existe également d'autres valeurs possibles comme par exemple **processing-instruction()**.

Nom	Description
nom du nœud	oriente la recherche vers le nœud dont le nom a explicitement été spécifié
*	oriente la recherche vers tous les nœuds
node()	oriente la recherche vers tous les types de nœuds (éléments, commentaires, attributs, etc.)
text()	oriente la recherche vers les nœuds de type texte
comment()	oriente la recherche vers les nœuds de type commentaire

Les chemins absolus

Dans notre premier exemple, le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- Etape 1 : descendre au nœud "**repertoire**".
- Etape 2 : descendre au nœud "**personne**".
- Etape 3 : descendre au nœud "**adresse**".
- Etape 4 : descendre au nœud "**pays**".

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- Etape 1 : `child::repertoire`.
- Etape 2 : `child::personne`.
- Etape 3 : `child::adresse`.
- Etape 4 : `child::pays`.

Ce qui nous donne :

`/child::repertoire/child::personne/child::adresse/child::pays`

Il est possible de simplifier l'écriture de cette expression =

`/repertoire/personne/adresse/pays`

L'expression XPath permettant de **trouver tous les commentaires de notre document XML**.

Dans ce nouvel exemple, une seule étape est en réalité nécessaire et consiste à sélectionner tous les descendants du nœud **racine** qui sont des commentaires.

Tentons maintenant de traduire cette étape sous la forme d'expressions XPath :

- On sélectionne tous les descendants avec l'expression **descendant**.
- On filtre les commentaires avec l'expression **comment()**.

`/descendant::comment()`

Les chemins relatifs

Notre point de départ sera le nœud "**telephones**". Une fois de plus, le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- Etape 1 : remonter au nœud frère "**adresse**".
- Etape 2 : descendre au nœud "**pays**".

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- Etape 1 : `preceding-sibling::adresse`.
- Etape 2 : `pays`.

Ce qui nous donne :

`preceding-sibling::adresse/pays`

L'expression `/descendant-or-self::node()`

Dans nos expressions XPath, il est possible de remplacer l'expression `"/descendant-or-self::node()"` par `"//"`.

Ainsi, l'expression :

`/descendant-or-self::node()/pays` peut être simplifiée par : `//pays`

L'expression `self::node()`

Notre deuxième abréviation va nous permettre de remplacer l'expression `"self::node()"` par `"."`.

Ainsi, l'expression :

`/repertoire/personne/self::node()` peut être simplifiée par : `/repertoire/personne/.`

L'expression `parent::node()`

Notre dernière abréviation va nous permettre de remplacer l'expression `"parent::node()"` par `".."`.

Les prédicats

Nom du prédicat	Description
attribute	permet d'affiner la recherche en fonction d'un attribut
count()	permet de compter le nombre de nœuds
last()	permet de sélectionner le dernier nœud d'une liste
position()	permet d'affiner la recherche en fonction de la position d'un nœud

A noter : il existe également d'autres valeurs possibles comme par exemple **name()**, **id()** ou encore **string-length()**. Je les ai volontairement retirées du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.

Un prédicat peut également contenir une expression XPath correspondant à une étape.

Quelques exemples

Pour les exemples, nous allons continuer de nous appuyer sur le même document XML :

Premier exemple

Dans notre premier exemple, le but va être de récupérer le nœud contenant le **numéro de téléphone fixe** de John DOE. Bien évidemment, il existe plusieurs façons d'y arriver. Je vous propose d'utiliser celle qui pousse le moins à réfléchir : nous allons sélectionner tous les descendants du nœud racine et filtrer sur la valeur de l'attribut **type**. Ce qui nous donne :

```
/descendant::*[attribute::type="fixe"]
```

Bien évidemment, cette méthode est à proscrire car elle peut avoir de nombreux effets de bord. Il est possible de procéder autrement en précisant le chemin complet :

```
/repertoire/personne/telephones/telephone[attribute::type="fixe"]
```

Terminons ce premier exemple en sélectionnant les numéros de téléphones qui ne sont pas des numéros de téléphones fixes. Une fois de plus, il existe plusieurs façons de procéder. La première, qui *a priori* est la plus simple, consiste à remplacer dans notre expression précédente l'opérateur d'égalité "=" par l'opérateur de non égalité "!=" :

```
/repertoire/personne/telephones/telephone[attribute::type!="fixe"]
```

Une autre méthode consiste à utiliser la fonction **not()** :

```
/repertoire/personne/telephones/telephone[not(attribute::type="fixe")]
```

A noter : la double négation nous fait revenir à notre point de départ. En effet, les 2 expressions suivantes sont équivalentes :

```
/repertoire/personne/telephones/telephone[not(attribute::type!="fixe")]
```

```
/repertoire/personne/telephones/telephone[attribute::type="fixe"]
```

Deuxième exemple

Après avoir manipulé les attributs, je vous propose maintenant de manipuler les positions. Ainsi, notre deuxième exemple consiste à sélectionner le **premier numéro de téléphone** de John DOE. Commençons par détailler les étapes en français :

- Etape 1 : descendre au nœud "**repertoire**".
- Etape 2 : descendre au nœud "**personne**".
- Etape 3 : descendre au nœud "**telephones**".
- Etape 4 : sélectionner le premier nœud "**telephone**".

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- Etape 1 : repertoire.
- Etape 2 : personne.
- Etape 3 : telephones.
- Etape 4 : telephone[position()=1].

Ce qui nous donne :

[/repertoire/personne/telephones/telephone\[position\(\)=1\]](#)

Si l'on souhaite maintenant sélectionner le dernier nœud "téléphone" de la liste, on modifiera l'expression de la manière suivante :

[/repertoire/personne/telephones/telephone\[last\(\)\]](#)

Quelques abréviations

Comme pour les axes, nous n'allons voir ici qu'une seule abréviation et elle concerne le prédicat **attribute** qu'il est possible de remplacer par le symbole "@". Ainsi, l'expression :

[/repertoire/personne/telephones/telephone\[attribute::type="fixe"\]](#)

devient :

[/repertoire/personne/telephones/telephone\[@type="fixe"\]](#)

Un exemple avec EditiX

Pour conclure ce chapitre, je vous propose de voir comment **exécuter une expression XPath avec EditiX**.

Le document XML


```

1  <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2
3  <repertoire>
4      <!-- John DOE -->
5      <personne sexe="masculin">
6          <nom>DOE</nom>
7          <prenom>John</prenom>
8          <adresse>
9              <numero>7</numero>
10             <voie type="impasse">impasse du chemin</voie>
11             <codePostal>75015</codePostal>
12             <ville>PARIS</ville>
13             <pays>FRANCE</pays>
14         </adresse>
15         <telephones>
16             <telephone type="fixe">01 02 03 04 05</telephone>
17             <telephone type="portable">06 07 08 09 10</telephone>
18         </telephones>
19         <emails>
20             <email type="personnel">john.doe@wanadoo.fr</email>
21             <email type="professionnel">john.doe@societe.com</email>
22         </emails>
23     </personne>
24
25     <!-- Marie POPPINS -->
26     <personne sexe="feminin">
27         <nom>POPPINS</nom>
28         <prenom>Marie</prenom>
29         <adresse>
30             <numero>28</numero>
31             <voie type="avenue">avenue de la république</voie>
32             <codePostal>13005</codePostal>
33             <ville>MARSEILLE</ville>
34             <pays>FRANCE</pays>
35         </adresse>
36         <telephones>
37             <telephone type="bureau">04 05 06 07 08</telephone>
38         </telephones>
39         <emails>
40             <email type="professionnel">contact@poppins.fr</email>
41         </emails>
42     </personne>
43 </repertoire>
44

```

La vue XPath

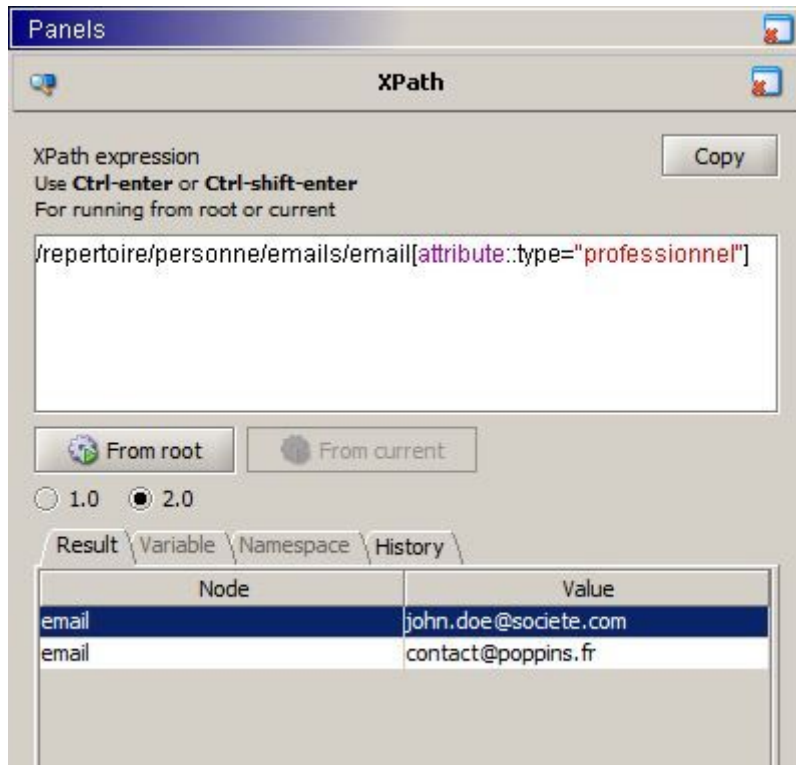
Afin de pouvoir exécuter des expressions XPath, nous allons devoir afficher la vue dédiée au sein de EditiX. Pour ce faire, vous pouvez sélectionner dans la barre de menu **XML** puis **XPath view** ou encore utiliser le raccourci clavier Ctrl + Shift + 4.

Exécuter une requête

Dans cet ultime exemple, nous allons sélectionner les nœuds contenant des adresses e-mails professionnelles grâce à l'expression suivante :

```
/repertoire/personne/emails/email[attribute::type="professionnel"]
```

En théorie, nous devrions avoir 2 nœuds sélectionnés. Vérifions tout de suite :



TP : des expressions XPath dans un répertoire

Le document XML

Voici les informations que l'on connaît pour chaque personne :

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

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <repertoire>
3   <!-- John DOE -->
4   <personne sexe="masculin">
5     <nom>DOE</nom>
6     <prenom>John</prenom>
7     <adresse>
8       <numero>7</numero>
9       <voie type="impasse">impasse du chemin</voie>
10      <codePostal>75015</codePostal>
11      <ville>PARIS</ville>
12      <pays>FRANCE</pays>
13    </adresse>
14    <telephones>
15      <telephone type="fixe">01 02 03 04 05</telephone>
16      <telephone type="portable">06 07 08 09 10</telephone>
17    </telephones>
18    <emails>
19      <email type="personnel">john.doe@wanadoo.fr</email>
20      <email type="professionnel">john.doe@societe.com</email>
21    </emails>
22  </personne>
23
24  <!-- Marie POPPINS -->
25  <personne sexe="feminin">
26    <nom>POPPINS</nom>
27    <prenom>Marie</prenom>
28    <adresse>
29      <numero>28</numero>
30      <voie type="avenue">avenue de la république</voie>
31      <codePostal>13005</codePostal>
32      <ville>MARSEILLE</ville>
33      <pays>FRANCE</pays>
34    </adresse>
35    <telephones>
36      <telephone type="professionnel">04 05 06 07 08</telephone>
37    </telephones>
38    <emails>
39      <email type="professionnel">contact@poppins.fr</email>
40    </emails>
41  </personne>
42
43  <!-- Batte MAN -->
44  <personne sexe="masculin">
45    <nom>MAN</nom>
46    <prenom>Batte</prenom>
47    <adresse>
48      <numero>24</numero>
49      <voie type="avenue">impasse des héros</voie>
50      <codePostal>11004</codePostal>
51      <ville>GOTHAM CITY</ville>
52      <pays>USA</pays>
53    </adresse>
54    <telephones>
55      <telephone type="professionnel">01 03 05 07 09</telephone>
56    </telephones>
57  </personne>
58 </repertoire>
59
```

Les expressions à écrire

Voici donc la liste des expressions XPath à écrire :

- Sélectionner tous les nœuds descendants du deuxième nœud **"personne"**.
- Sélectionner le nœud **"personne"** correspondant à l'individu ayant au moins 2 numéros de téléphone.
- Sélectionner tous les nœuds **"personne"**.
- Sélectionner le deuxième nœud **"personne"** dont le pays de domiciliation est la **France** .
- Sélectionner tous les nœuds **"personne"** de sexe **masculin** le pays de domiciliation est les **Etats-Unis**.

Expression n°1

Le but de cette première expression était de sélectionner tous les nœuds descendants du deuxième nœud **"personne"** :

```
1 /repertoire/personne[position()=2]/descendant::*
```

Expression n°2

Le but de cette expression était de sélectionner le nœud **"personne"** correspondant à un individu ayant au moins 2 numéros de téléphone :

```
1 /repertoire/personne[count(telephones/telephone) > 1]
```

Expression n°3

Le but de cette troisième expression était de sélectionner tous les nœuds **"personne"** :

```
1 /repertoire/personne
```

ou encore :

```
1 //personne
```

Expression n°4

Le but de cette expression était de sélectionner le deuxième nœud **"personne"** dont le pays de domiciliation est la **France** :

```
1 /repertoire/personne[adresse/pays="FRANCE"][position()=2]
```

Expression n°5

Le but de la dernière expression était de sélectionner tous les nœuds **"personne"** de sexe **masculin** le pays de domiciliation est les **Etats-Unis** :

```
1 /repertoire/personne[@sexe="masculin"][adresse/pays="USA"]
```