

## 5. Schémas XML

### 5.1 Introduction

#### 5.1.1 Pourquoi des schémas XML ?

Les spécifications **XML** définissent ce qu'est un document **bien formé** et **valide**.

Le premier point signifie tout simplement que le document respecte la syntaxe **XML**, alors que le second impose qu'il possède une **DTD** et qu'il en respecte les contraintes.

Toutefois, certaines applications peuvent nécessiter la définition de structures plus riches et de contraintes différentes de celles qui sont à la portée d'une **DTD**.

D'autre part, il paraîtrait souhaitable que les contraintes soient exprimées à l'aide d'une syntaxe **XML** (*contrairement aux DTD*) ce qui permettrait de les traiter avec des outils standards.

Au regard de ces considérations, le **W3C** a jugé nécessaire de concevoir un langage simple et facile à utiliser, qui soit plus expressif qu'une **DTD**, qui utilise une syntaxe **XML** et soit compatible avec le reste de l'éco-système (*Namespaces, HTML, DOM, RDF, XSLT, ...*).

#### 5.1.2 Les spécifications XML Schema

Les spécifications de **XML Schema** ont été découpées en trois documents :

**XML Schema Part 0 : Primer** (Rec. W3C - 2ème éd. Octobre 2004)

Un document non normatif qui décrit le langage afin d'en faciliter la compréhension et d'aider au développement de schémas.

 Consulter le document

<http://www.w3.org/TR/xmlschema-0/>

**XML Schema Part 1 : Structures** (Rec. W3C - 2ème éd. Octobre 2004)

Précise ce qu'est un schéma **XML**, référence les éléments permettant de construire des schémas, et définit comment appliquer des schémas aux documents **XML**.

 Consulter le document

<http://www.w3.org/TR/xmlschema-1/>

**XML Schema Part 2 : Datatypes** (Rec. W3C - 2ème éd. Octobre 2004)

Présente les types de données qui peuvent être utilisées dans un schéma **XML**. Ces données peuvent correspondre à la valeur d'un noeud texte ou d'un attribut. Peut être utilisée hors du contexte "*schémas*".

 Consulter le document

<http://www.w3.org/TR/xmlschema-2/>

#### 5.1.3 Fiche d'identité

**XML Schema** est une application **XML**. L'extrait ci-dessous en fait apparaître l'**URI** public et l'**URL** privée permettant de localiser la **DTD**, ainsi que l'**URI** d'espace de noms et un préfixe usuel.

```
<?xml version="1.0"?>
<!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN"
        "http://www.w3.org/2001/XMLSchema.dtd">
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    .
    .
    .
</xs:schema>
```

 Consulter la DTD des schémas

<http://www.w3.org/2001/XMLSchema.dtd>

**N.B.** Il existe deux préfixes couramment associés aux schémas : "**xs**" et "**xsd**".

Outre la **DTD** dont l'**URL** est mentionnée ci-dessus, il existe également un schéma de **XML Schema**.

☞ Accéder au schéma des schémas

<http://www.w3.org/2001/XMLSchema.xsd>

#### 5.1.4 Associer un schéma à un document

L'association d'un schéma à un document **XML** se fait à l'aide d'un attribut spécial, de préférence affecté à l'élément racine du document.

Cet attribut est rattaché à un espace de noms particulier, et mentionne l'**URL** du schéma associé au document :

```
<page id="xml/xsd/intro/usage.xml"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="syntax/cours.xsd">
  .
  .
  .
</page>
```

Cet exemple correspond au cas le plus simple, celui où le document n'utilise pas d'espaces de noms pour son vocabulaire propre.

#### > Espaces de noms

Dans le cas contraire, l'attribut associant le schéma au document doit indiquer quel est l'espace de noms du vocabulaire auquel correspond le schéma :

```
<dm:page id="xml/xsd/intro/usage.xml"
          xmlns:dm="http://tic.ec-lyon.fr/~muller/2002/cours"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://tic.ec-lyon.fr/~muller/2002/cours syntax/cours.xsd">
  .
  .
  .
</dm:page>
```

On remarquera que l'espace de noms est logiquement identifié par son **URI** et non par un quelconque préfixe...

**N.B.** Dans le cas où le document utilise plusieurs vocabulaires, avec chacun son espace de noms et un schéma associé, l'attribut "*schemaLocation*" prend pour valeur une liste de paires (*séparées par des blancs*) formées par un **URI** d'espace de noms suivi par l'**URL** du schéma correspondant.

#### > Des éléments indicatifs

Les spécifications disent que ces informations doivent être considérées comme indicatives.

Toute application travaillant avec un document et un schéma associé, doit offrir à l'utilisateur des méthodes alternatives (*fichier de configuration, ligne de commande, ...*) qui lui permettront d'utiliser un schéma autre que celui mentionné au sein du document.

#### 5.1.5 A quoi sert un schéma XML ?

Un schéma donne un ensemble de règles décrivant la structure et le contenu d'une classe de documents **XML**. Ces informations sont susceptibles d'être exploitées à bien des fins différentes, rapidement évoquées ci-dessous.

#### > Validation

La validation est l'application la plus courante des schémas. Elle consiste à vérifier que le document est bien conforme à ce qui est décrit dans le schéma.

Cette validation vérifie la conformité structurelle du document (*arborescence des éléments et attributs*) ainsi que le type des données (*valeurs textuelles*) qui doivent correspondre à ce qui est autorisé par le schéma.

La validation permet de vérifier qu'un document reçu par une application est conforme à ce qu'elle attend, de manière à s'assurer que les traitements qui seront effectués ne soient pas erronés.

## > Documentation

L'un des avantages essentiels d'un document **XML** est qu'il est lisible par des humains. Cette remarque s'applique bien sûr aux schémas.

Un schéma est une excellente méthode pour documenter un vocabulaire **XML** de manière précise, et impossible à atteindre en plein texte.

Une fois écrit, il suffit de tenter la validation d'une série de documents existants pour vérifier si la *"documentation"* est à jour !

Sachant qu'un schéma est lui-même un document **XML**, il est d'autre part tout à fait envisageable de le visualiser ou de le transformer à l'aide d'outils génériques (*éditeur d'arbre XML, XSLT, ...*) afin de le rendre encore plus lisible par un humain (*peut-être moins informaticien que d'autres*).

## > Requêtage

Certaines applications travaillent en extrayant des parties de documents pour leur appliquer un traitement spécifique (*cf. XQuery ou XSLT via XPath...*).

A l'heure actuelle, ces applications sont totalement génériques, et travaillent *"en aveugle"*, sans connaître a priori la structure du document analysé.

Si la généricité fait évidemment la force de telles applications, il n'en demeure pas moins que certaines opérations comme le tri (*alphabétique, numérique, par date, ...*) pourraient être bien optimisées par la connaissance de la structure du document et de la nature des données.

Ces informations sont contenues dans les schémas. C'est la raison pour laquelle les spécifications en cours de gestation (*XPath 2.0, XSLT 2.0, XQuery 1.0*) s'appuieront sur les schémas quand ils seront disponibles.

## > Edition

L'information disponible dans un schéma peut permettre à un éditeur générique intelligent de s'adapter à une application particulière et de ne proposer à l'utilisateur que les choix qui sont autorisés à tout instant en fonction du contexte (*éléments structurels*).

De plus, l'éditeur pourra également vérifier la valeur textuelle des noeuds, pour s'assurer qu'elle est compatible avec les types de données préconisés par le schéma.

## 5.2 Principes

### 5.2.1 Exemple de schéma XML

-----  
Considérons une application simple, qui liste l'ensemble des étudiants qui assistent à un cours :

```
<classe>
  <promo>2004</promo>
  <étudiant>
    <prénom>Raymond</prénom>
    <nom>Deubaze</nom>
  </étudiant>
  <étudiant>
    <prénom>Ginette</prénom>
    <nom>Ringard</nom>
  </étudiant>
  ...
</classe>
```

L'extrait de schéma suivant définit la liste des éléments de cette application :

```
<xs:element name="classe" type="type-classe"/>
<xs:element name="promo" type="xs:string"/>
<xs:element name="étudiant" type="type-etudiant"/>
<xs:element name="prénom" type="xs:string"/>
<xs:element name="nom" type="xs:string"/>
```

On remarque que les éléments texte sont déclarés comme étant du type `"xs:string"`. La présence du préfixe correspondant à l'espace de noms de **XML Schema** n'est pas fortuite : il indique qu'il s'agit d'un type prédéfini.

Le type des éléments à contenu élémentaire (*element content*) doit être défini par ailleurs :

- L'élément `"étudiant"` comportera un `"prénom"` et un `"nom"`, dans cet ordre :

```
<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

- L'élément classe comportera un élément **promo** suivi par un ou plusieurs éléments `"étudiant"` :

```
<xs:complexType name="type-classe">
  <xs:sequence>
    <xs:element ref="promo"/>
    <xs:element ref="étudiant" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/exemple-classe.html>]

## 5.2.2 Modèles de contenu courants

### > Éléments texte

Le type des éléments à contenu purement textuel est dit **simple** :

```
<nom>Deubaze</nom>
```

**XML Schema** propose une bibliothèque de types simples réutilisables (*chaînes de caractères, nombres, dates, ...*) :

```
<xs:element name="nom" type="xs:string"/>
```

De plus, il est possible de définir ses propres types simples par dérivation à partir de types existants (*extension, restriction ou union*).

L'exemple ci-dessous impose que la valeur de l'élément `"nom"` soit un mot en minuscules, de 2 caractères au minimum, avec une majuscule initiale :

```
<xs:element name="nom" type="type-nom"/>
<xs:simpleType name="type-nom">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z] [a-z] +"/>
  </xs:restriction>
</xs:simpleType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/text-content-exemple.html>]

### > Éléments texte avec attributs

Les éléments à contenu textuel pur munis d'attributs constituent un cas particulier :

```
<prénom usuel="oui">Raymond</prénom>
```

**XML Schema** parle ici d'éléments de **type complexe à contenu simple**.

On définit ce type d'éléments par dérivation à partir d'un type simple, ou d'un élément complexe à contenu simple, en ajoutant (*extension*) les attributs :

```
<xs:element name="prénom" type="type-prénom"/>
<xs:complexType name="type-prénom">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="usuel" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/text-attribute-content-exemple.html>]

Noter au passage que les attributs possèdent eux aussi un modèle de contenu simple référencé à l'aide de l'attribut *"type"* de l'élément *xs:"attribute"*.

### > Éléments à contenu élémentaire

Les éléments ne contenant que des sous-éléments sont dits à contenu élémentaire (*element content*) :

```
<étudiant>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
</étudiant>
```

**XML Schema** considère que le type de tous les éléments à contenu autre que purement textuel est **complexe**. Il n'y a pas de types complexes prédéfinis.

Contrairement à ce qui se passe pour les nouveaux types simples, qui ne peuvent être créés mais uniquement obtenus par dérivation à partir de types existants (*prédéfinis ou non*), les types complexes peuvent être soit créés, soit dérivés.

### Exemple de création d'un nouveau type complexe :

La création d'un type complexe se fait essentiellement en donnant la liste (*xs:sequence*) des sous-éléments autorisés. Cette liste peut éventuellement prendre en compte des alternatives mutuellement exclusives (*xs:choice*).

L'exemple ci-dessous définit un élément *"étudiant"* qui doit contenir exactement un *"nom"* et éventuellement plusieurs *"prénom"*, dans un ordre quelconque.

```
<xs:element name="étudiant" type="type-étudiant"/>
<xs:complexType name="type-étudiant">
  <xs:choice>
    <xs:sequence>
      <xs:element ref="nom"/>
      <xs:element ref="prénom" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element ref="prénom" maxOccurs="unbounded"/>
      <xs:element ref="nom"/>
      <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/element-content-exemple.html>]

### Exemple de dérivation à partir d'un type existant :

Soit à autoriser un attribut *"civilité"* pour l'élément *"étudiant"* :

```
<étudiant civilité="M.">
  <prénom>Jean</prénom>
  <nom>Aymard</nom>
</étudiant>
```

L'exemple ci-dessous s'appuie sur le type *"étudiant"* préalablement défini en ajoutant l'attribut par extension :

```
<xs:element name="étudiant" type="etudiant-civilise"/>
<xs:complexType name="etudiant-civilise">
  <xs:complexContent>
    <xs:extension base="type-etudiant">
      <xs:attribute name="civilité" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/element-content-exemple2.html>]

### > Éléments à contenu mixte

Les éléments à contenu mixte peuvent contenir aussi bien du texte que des sous-éléments :

```
<prénom>Stanislas 1<sup>er</sup></prénom>
```

Les éléments à contenu mixte sont déclarés comme des éléments complexes, avec un attribut supplémentaire indiquant la présence éventuelle de texte :

```
<xs:element name="prénom" type="type-prénom"/>
<xs:element name="sup" type="xs:string"/>
<xs:complexType name="type-prénom" mixed="true">
  <xs:sequence>
    <xs:element ref="sup" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/mixed-content-exemple.html>]

Remarquer que ceci revient à traiter les éléments à contenu mixte comme un cas particulier des éléments à contenu élémentaire.

Il en résulte que les possibilités de contraintes sur la structure du document (*ordre, imbrication et cardinalité des éléments*) existent de la même manière pour les éléments à contenu mixte. Il est par contre tout à fait impossible de contrôler finement la position ou la nature du texte présent entre les éléments.

### > Éléments vides

Les éléments vides ne peuvent contenir ni de texte, ni d'autres éléments, mais peuvent éventuellement être porteurs d'attributs :

```
<promo année="2005"/>
<étudiant>
  <prénom>Snoopy</prénom>
  <mascotte/>
</étudiant>
```

Pour définir le type de contenu d'un élément vide, on peut spécifier un contenu complexe sans sous-éléments, ou alors un contenu simple de longueur nulle.

#### Élément vide - modèle complexe :

```
<xs:complexType name="empty">
  <xs:sequence/>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/empty-content-exemple1.html>]

#### Élément vide - modèle simple :

```
<xs:simpleType name="empty">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/empty-content-exemple2.html>]

**N.B.** Un élément vide ne peut comporter ni de noeud texte, ni de sous-élément. Il peut par contre contenir des commentaires ainsi que des instructions de traitement.

### 5.2.3 Création de types par dérivation

Les exemples précédents ont montré qu'il était possible de créer de nouveaux types de contenu par extension ou dérivation d'un type existant :

```
<xs:complexType name="type-mascotte">
  <xs:simpleContent>
    <xs:extension base="empty">
      <xs:attribute name="source" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Exemple d'extension :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/derivation-exemple1.html>]

Les possibilités de dérivation sont la **restriction**, la **liste**, et l'**union**.

La restriction est un peu particulière, puisqu'elle permet de limiter le champ des possibles en agissant sur certaines caractéristiques du type modifié (*longueur d'une chaîne, valeur d'un nombre, ...*). Les paramètres sur lesquels il est possible d'agir par restriction sont appelés des **facettes**.

La déclaration d'un élément vide sur la base d'un contenu simple est un exemple de dérivation par restriction agissant sur la facette "*maxLength*" du type prédéfini "*xs:string*" :

```
<xs:simpleType name="empty">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/derivation-exemple2.html>]

### 5.2.4 Définitions locales

Les définitions d'éléments et les définitions de type vues en exemple jusqu'à présent déclaraient des éléments et des types identifiés par un nom (*cf. attribut name*) d'une portée **globale**, accessibles depuis l'ensemble du schéma :

```
<xs:element name="étudiant" type="type-etudiant"/>
<xs:element name="prénom" type="xs:string"/>
<xs:element name="nom" type="xs:string"/>

<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

**N.B.** Les définitions globales servent souvent à constituer des bibliothèques de types réutilisables.

**XML Schema** offre également la possibilité de définir les éléments et les types localement, sans leur donner de nom.

#### Définition locale de type :

L'exemple ci-dessous montre l'élément *"étudiant"* dont le type (*non nommé*) est défini localement :

```
<xs:element name="prénom" type="xs:string"/>
<xs:element name="nom" type="xs:string"/>

<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="prénom"/>
      <xs:element ref="nom"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

#### Définition locale d'élément :

Comme le montre l'exemple ci-dessous, les éléments peuvent eux-mêmes être définis localement (*plutôt que référencés*) :

```
<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

#### Aller au bout de la logique :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/localdefs-exemple.html>]

On parlera de **schémas plats** (*cf. structure de l'arbre XML*) lorsque ceux-ci sont fortement axés sur des définitions globales, et de **schémas profonds** lorsqu'ils font surtout usage de définitions locales.



### 5.2.5 Eléments fonction du contexte

Les spécifications de **XML** disent qu'une application comporte un certain nombre d'éléments identifiés par leur *"type"* (i.e. *par leur nom*), éventuellement discriminés par des espaces de noms.

Or les définitions locales, permettent de facto l'usage d'éléments dont la définition varie en fonction du contexte :

```
<cours>
  <nom>Technologies XML</nom>
  <étudiant>
    <nom>Deubaze</nom>
  </étudiant>
</cours>
```

Dans l'application ci-dessus, le *"nom"* du cours sera une chaîne générique, tandis que le *"nom"* de l'étudiant sera limité à un seul mot commençant par une majuscule :

```
<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element name="nom">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="[A-Z] [a-z] +"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="type-classe">
  <xs:sequence>
    <xs:element name="nom" type="xs:string"/>
    <xs:element ref="promo"/>
    <xs:element ref="étudiant" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/typecontext-exemple.html>

**N.B.** Cette approche était courante pour les attributs (*il est coutumier d'avoir des attributs portant le même nom attachés à des éléments différents*), mais est relativement nouvelle pour les éléments.

Elle est généralisée par l'application **XML Schema** elle-même qui réutilise dans des contextes différents des éléments avec des significations voisines mais pas toujours identiques.

## 5.3 Les types simples prédéfinis

### 5.3.1 Introduction

Avant de s'appesantir sur l'ensemble des types prédéfinis, et afin de bien comprendre certains aspects, il est nécessaire de détailler le processus allant de la lecture du document brut à l'obtention d'un **arbre XML** en mémoire.

#### > Espace physique

On conviendra d'appeler **espace physique** le niveau correspondant au texte brut tel qu'il apparaît dans un ou plusieurs documents physiques.

**N.B.** Lorsque les documents sont conservés sur un support quelconque (*disque, bande, mémoire flash, ...*) l'espace physique pourrait être appelé **espace de stockage**. Ce terme ne peut toutefois pas être employé dans le cas général pour tenir compte des documents purement dynamiques.

### > Espace Unicode

La recommandation **XML 1.0** indique que la façon dont les documents sont stockés, segmentés ou transmis n'est pas significative pour les applications. Les parseurs conformes à **XML** doivent avant tout convertir la forme physique d'un document pour obtenir une version **normalisée**.

Une première étape consiste à convertir tous les caractères en **Unicode**, et ceci quel que soit l'encodage utilisé dans l'espace physique. Cette transformation concerne l'ensemble du document (*noms des éléments, des attributs, contenu textuel, ...*).

On conviendra dans ce cours d'appeler l'état obtenu après prise en compte des divers documents physiques et conversion des codes caractères l'**espace Unicode**.

### > Espace normalisé

Dans un second temps, tous les espaces (*TAB* - #x09, *NL* - #x0A, *CR* - #x0D) sont convertis en blancs (#x20). Cette transformation est appelée la **normalisation**. Le nombre de caractères n'est pas modifié par cette opération.

Un analyseur validant normalise (*aux fins de validation*) tous les types de données prédéfinis sauf *"xs:string"*.

On dira après cette étape qu'une information se trouve dans l'**espace normalisé**.

### > Espace lexical

Avant de pouvoir valider le document, les recommandations définissent une seconde transformation appelée **compactage** qui consiste à retirer les blancs initiaux et finaux, et à remplacer toutes les séquences de blancs consécutifs par un blanc unique.

Le **compactage** modifie le nombre de caractères. Il s'applique à tous les types de données prédéfinis sauf *"xs:string"* et *"xs:normalizedString"*.

La valeur obtenue après compactage éventuel appartient alors à l'**espace lexical**.

### > Espace des valeurs

Dans l'espace lexical, on ne dispose encore que d'une représentation textuelle d'une valeur logique dont le sens est défini par le type qui lui est associé (*chaîne, nombre, date, ...*).

L'ensemble constitué des valeurs logiques, après prise en compte du type s'appelle l'**espace des valeurs**.

### > Remarques

Chaque type de données a son propre espace lexical (*ensemble des représentations textuelles autorisées*) et son propre espace des valeurs. La correspondance entre ces deux espaces est spécifique à chaque type.

Il arrive souvent qu'une même valeur puisse avoir plusieurs représentations lexicales (*cf. 1, +1, 10e-1, ...*).

Cette subtilité est importante lors de comparaisons ou d'opérations de tri. En effet les valeurs associées aux représentations lexicales *"1"* et *"1.0"* seront les mêmes si leur type est *"xs:float"* mais différentes si c'est *"xs:string"*.

Enfin, il est important de noter que les informations transmises à une application sont celles situées dans un **espace normalisé** légèrement différent de celui présenté ici (*retours à la ligne #x0A, et compactage des valeurs d'attributs*). Toutes les opérations suivantes sont uniquement effectuées à des fins de validation et ne concernent que les analyseurs validants.

## 5.3.2 Chaînes brutes

-----

### > xs:string

Le type *"xs:string"* est le seul type prédéfini pour lequel il n'y a pas de normalisation (*et par suite, par de compactage non plus*) avant validation. L'**espace des valeurs** est confondu avec l'**espace Unicode**.

En théorie cet aspect ne concerne que la **validation** et ne préjuge donc pas de ce qui est transmis à l'application...

### > `xs:normalizedString`

Comme son nom l'indique, une donnée du type "`xs:normalizedString`" est normalisée mais non compactée. L'**espace des valeurs** est confondu avec l'**espace normalisé**.

La seule différence avec "`xs:string`" est que les séparateurs non blancs sont transformés en blancs avant validation.

**N.B.** Il s'agit du seul type prédéfini normalisé mais non compacté.

### 5.3.3 Chaînes de caractères

---

#### > `xs:token`

Ce type est une version compactée de "`xs:normalizedString`". Les blancs initiaux et finaux ont été supprimés, et les blancs consécutifs réduits à une seule occurrence. L'**espace des valeurs** est confondu avec l'**espace lexical**.

```
<xs:attribute name="titre" type="xs:token"/>
```

```
<cours titre="XML, standards et applications"/>
```

**N.B.** En toute rigueur, et contrairement à ce que semble indiquer le nom de ce type, le résultat obtenu dans l'espace des valeurs ressemble plus à une liste de mots séparés par un espace, qu'à une unité lexicale unique...

#### > `xs:NMTOKEN`

Ce type correspond à un **nom XML** sans la restriction sur la nature du premier caractère. Seuls sont autorisés les caractères alphanumériques, et les caractères "*souligné*" ( \_ ), "*tiret*" ( - ), "*point*" ( . ) et "*double-point*" ( : ). Les blancs sont interdits.

```
<xs:attribute name="sigle" type="xs:NMTOKEN"/>
```

```
<produit sigle="2B3"/>
```

"`xs:NMTOKEN`" est dérivé de "`xs:token`" par restriction.

#### > `xs:NMTOKENS`

Une information du type "`xs:NMTOKENS`" est une liste de données du type "`xs:NMTOKEN`" séparées par des espaces.

```
<xs:attribute name="hosts" type="xs:NMTOKENS"/>
```

```
<Allow hosts="156.18.10.1 localhost tic01.tic.ec-lyon.fr"/>
```

"`xs:NMTOKENS`" est dérivé de "`xs:NMTOKEN`" par liste.

#### > `xs:Name`

Il s'agit ici d'un **nom XML** avec la restriction sur la nature du premier caractère qui ne peut être qu'alphabétique.

```
<xs:element name="prénom" type="xs:Name"/>
```

```
<prénom>Jean-Pierre</prénom>
```

"`xs:Name`" est dérivé de "`xs:token`" par restriction.

#### > `xs:NCName`

"`NCName`" signifie "*Non Colonized Name*" (nom sans double-points). Ce type correspond à "`xs:Name`" sans le droit au "*double-point*" ( : ).

Il s'agit du type le proche de ce que l'on pourrait appeler un nom (*quoique sans espaces*) dans le langage courant, ou un nom de variable en programmation, bien que son usage en tant que tel autorise encore des caractères tels le "*souligné*" ( \_ ), le "*tiret*" ( - ) et le "*point*" ( . ).

```
<xs:element name="HostName" type="xs:NCName"/>
```

```
<HostName>tic01.tic.ec-lyon.fr</HostName>
```

"*xs:NCName*" est dérivé de "*xs:Name*" par restriction.

#### > *xs:QName*

"*QName*" signifie "*Qualified Name*". Il s'agit d'un **nom XML** associé à son **espace de noms**. Au niveau de l'espace lexical est permis un unique caractère "*double-point*" ( : ) non situé en position initiale.

L'espace des valeurs est très particulier, puisque la valeur associée à un nom qualifié est constituée d'un couple comportant l'**URI de l'espace de noms** et la chaîne suivant le "*double-point*".

Dans le cas suivant (*déclaration d'un attribut type du type QName*) :

```
<xs:attribute name="type" type="xs:QName"/>
```

```
<variable type="xs:string"/>
```

et dans la mesure où le préfixe "*xs*" a été déclaré de la manière habituelle, alors la chaîne "*xs:string*" (dans l'espace lexical) aura pour valeur (dans l'espace des valeurs) le couple ("*http://www.w3.org/2001/XMLSchema,string*").

Si le préfixe apparaissant dans l'espace lexical n'a pas été associé à un **URI** d'espace de noms, alors l'information ne sera pas validée.

**N.B.** En l'absence de préfixe, le terme correspondant à l'**URI** d'espace de noms dans l'espace des valeurs vaut "*NULL*".

"*xs:QName*" est un type primitif. Il n'est pas obtenu par dérivation.

### 5.3.4 Identifiants uniques

#### > *xs:ID*

L'espace lexical du type "*xs:ID*" est le même que celui du type "*xs:NCName*". En d'autres termes il s'agit là encore d'un **nom XML** privé du droit au "*double-point*" ( : ).

La différence se situe dans le fait que la valeur obtenue doit être unique au sein du document. Ceci est à rapprocher des attributs du type **ID** tels qu'il est possible de les déclarer à l'aide d'une **DTD** :

```
<xs:attribute name="insee" type="xs:ID"/>
```

```
<personne insee="I2631069210009">
  . . .
</personne>
```

"*xs:ID*" est dérivé de "*xs:NCName*" par restriction.

#### > *xs:IDREF*

Il s'agit là encore d'un type dont l'espace lexical est identique à celui de "*xs:NCName*".

La valeur obtenue doit correspondre à une valeur du type **xs:ID** trouvée ailleurs dans le même document.

```
<xs:attribute name="insee" type="xs:ID"/>
<xs:attribute name="prof" type="xs:IDREF"/>
```

```
<personne insee="I2631069210009">
  . . .
</personne>

<cours prof="I2631069210009">
  . . .
</cours>
```

"*xs:IDREF*" est dérivé de "*xs:NCName*" par restriction.

#### > *xs:IDREFS*

Une information du type "*xs:IDREFS*" est une liste de données du type "*xs:IDREF*" séparées par des espaces.

```
<xs:attribute name="élèves" type="xs:IDREFS"/>

<élève id="t1" nom="Deubaze" prénom="Raymond"/>
<élève id="t2" nom="Ringard" prénom="Ginette"/>
<élève id="t3" nom="Aymard" prénom="Jean"/>

<cours nom="XML, Standards et Applications" élèves="t1 t2 t3"/>
```

"*xs:IDREFS*" est dérivé de "*xs:IDREF*" par liste.

### 5.3.5 Types particuliers

#### > *xs:language*

"*xs:language*" est spécialement conçu pour accepter les noms de langues normalisés, tels que décrits par la **RFC 1766** "*Tags for the Identification of Languages*".

```
<xs:attribute name="lang" type="xs:language"/>
```

```
<text lang="en-US"/>
```

🔗 Consulter la RFC 1766

[<https://www.ietf.org/rfc/rfc1766.txt>]

🔗 Voir aussi la RFC 3066

[<https://www.ietf.org/rfc/rfc3066.txt>]

"*xs:language*" est dérivé de "*xs:token*" par restriction.

#### > *xs:anyURI*

Ce type est construit pour autoriser tous types d'**URI** tels que définis par la **RFC 2396** "*Uniform Resource Identifiers (URI): Generic Syntax*" et la **RFC 2732** "*Format for Literal IPv6 Addresses in URL's*".

```
<xs:attribute name="tdm" type="xs:anyURI"/>
```

```
<cours tdm="http://tic01.tic.ec-lyon.fr/~muller/cours/xml"/>
```

🔗 Consulter la RFC 2396

[<https://www.ietf.org/rfc/rfc2396.txt>]

🔗 Consulter la RFC 2732

[<https://www.ietf.org/rfc/rfc2732.txt>]

La particularité de ce type est que l'**espace des valeurs** et l'**espace lexical** ne sont pas confondus, pour prendre en compte le fait qu'un **URI** ne peut être exprimé qu'à l'aide de caractères **ASCII**.

Voici un exemple :

Espace lexical : `http://www.exemple.fr/Page d'entrée.html`

Espace des valeurs : `http://www.exemple.fr/Page%20d%27entr%e9e.html`

"*xs:anyURI*" est un type primitif. Il n'est pas obtenu par dérivation.

#### > *xs:boolean*

"*xs:boolean*" est un type primitif dont l'espace lexical est limité aux quatre expressions "*true*", "*false*", "*1*" et "*0*". L'espace des valeurs n'en contient que deux, les valeurs logiques "*vrai*" et "*faux*".

```
<xs:attribute name="hidden" type="xs:boolean"/>
```

```
<circle hidden="true"/>
```

"*xs:boolean*" est un type primitif. Il n'est pas obtenu par dérivation.

#### > *xs:hexBinary*

Il s'agit d'un type permettant la représentation de flux de données binaires arbitraires. La représentation lexicale d'un octet quelconque consiste à le coder sous la forme de deux caractères hexadécimaux.

```
<xs:element name="data" type="xs:hexBinary"/>
```

```
<data>3F80</data>
```

La valeur d'une donnée du type "*hexBinary*" est la séquence d'octets obtenue par décodage. A titre d'illustration la valeur correspondant à l'exemple ci-dessus est le nombre binaire "*0011 1111 1000 0000*".

"*xs:hexBinary*" est un type primitif. Il n'est pas obtenu par dérivation.

#### > *xs:base64Binary*

"*base64Binary*" est encore un type permettant de représenter des données binaires. La différence avec le précédent se situe au niveau du codage utilisé.

Le codage "*base64*" décrit par la **RFC 2045** représente un groupe de 6 bits en série par un caractère, ce qui correspond à 4 caractères pour 3 octets (*i.e.* 24 bits).

```
<xs:element name="data" type="xs:base64Binary"/>
```

```
<data>aGVsbG8gV29ybGQgIQ==</data>
```

La valeur d'une donnée du type "*base64Binary*" est la séquence d'octets obtenue par décodage. A titre d'illustration la valeur correspondant à l'exemple ci-dessus est la chaîne de caractères "*hello World !*".

 Consulter la RFC 2045

[<https://www.ietf.org/rfc/rfc2045.txt>]

"*xs:base64Binary*" est un type primitif. Il n'est pas obtenu par dérivation.

### 5.3.6 Nombres

#### > *xs:decimal*

Le type "*xs:decimal*" permet de représenter n'importe quel nombre décimal. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un signe "+" ou "-" initial, les chiffres de "0" à "9" et le "*point*" ( *.* ) qui est obligatoire.

Les zéros initiaux (*avant la virgule, représentée par le point*) et finaux (*après la virgule*) présents dans l'espace lexical ne se retrouvent évidemment pas dans l'espace des valeurs, purement numériques.

```
<xs:attribute name="prix" type="xs:decimal"/>
```

```
<article désignation="baguette" prix="0.60" monnaie="euro"/>
```

"*xs:decimal*" est un type primitif. Il n'est pas obtenu par dérivation.

### > xs:integer

Le type "*xs:integer*" permet de représenter n'importe quel nombre entier. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un signe "+" ou "-" initial, et les chiffres de "0" à "9".

Les zéros initiaux présents dans l'espace lexical ne se retrouvent évidemment pas dans l'espace des valeurs, purement numériques.

```
<xs:attribute name="promo" type="xs:integer"/>
<xs:attribute name="effectif" type="xs:integer"/>
```

```
<option sigle="TI" promo="2005" effectif="29"/>
```

"*xs:integer*" est obtenu par restriction à partir de "*xs:decimal*".

### > xs:nonPositiveInteger

Ce type permet de représenter l'ensemble des entiers négatifs ou nuls. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont le signe "-" initial (*obligatoire sauf pour la valeur 0*), et les chiffres de "0" à "9".

```
<xs:attribute name="profondeur" type="xs:nonPositiveInteger"/>
```

```
<plongée conditions="apnée" profondeur="-162" unités="m"/>
```

"*xs:nonPositiveInteger*" est obtenu par restriction à partir de "*xs:integer*".

### > xs:negativeInteger

Ce type permet de représenter l'ensemble des entiers strictement négatifs. Le nombre de digits n'est pas limité. La représentation lexical comporte un signe "-" obligatoire suivi par un ensemble de chiffres de "0" à "9".

```
<xs:attribute name="datation" type="xs:negativeInteger"/>
```

```
<fossile type="hominidé" nom="Lucy" datation="-3600000"/>
```

"*xs:negativeInteger*" est obtenu par restriction à partir de "*xs:nonPositiveInteger*".

### > xs:nonNegativeInteger

Ce type permet de représenter l'ensemble des entiers positifs ou nuls. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un éventuel signe "+" suivi par un ensemble de chiffres de "0" à "9".

```
<xs:element name="âge" type="xs:nonNegativeInteger"/>
```

```
<âge>22</âge>
```

"*xs:nonNegativeInteger*" est obtenu par restriction à partir de "*xs:integer*".

### > xs:positiveInteger

Ce type permet de représenter l'ensemble des entiers strictement positifs. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un éventuel signe "+" suivi par un ensemble de chiffres de "0" à "9".

```
<xs:element name="promo" type="xs:positiveInteger"/>
```



```
<promo>2005</promo>
```

"*xs:positiveInteger*" est obtenu par restriction à partir de "*xs:nonNegativeInteger*".

#### > *xs:long*

Contrairement aux types précédents dont le nombre de digits n'était pas limité, le type "*xs:long*" admet l'ensemble des entiers signés dont la représentation binaire tient sur un mot de 64 bits, soit les valeurs allant de -9223372036854775808 à +9223372036854775807 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:long*" est obtenu par restriction à partir de "*xs:integer*".

#### > *xs:int*

Sur le même modèle que le type "*xs:long*", "*xs:int*" correspond aux entiers signés dont la représentation binaire tient sur 32 bits, soit les valeurs allant de -2147483648 à +2147483647 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:int*" est obtenu par restriction à partir de "*xs:long*".

#### > *xs:short*

Dans la même logique, "*xs:short*" s'adresse aux entiers signés dont la représentation binaire correspond à des mots de 16 bits, soit des valeurs de -32768 à +32767 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:short*" est obtenu par restriction à partir de "*xs:int*".

#### > *xs:byte*

"*xs:byte*" limite la représentation des entiers signés à 8 bits, soit des valeurs de -128 à +127 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:byte*" est obtenu par restriction à partir de "*xs:short*".

#### > *xs:unsignedLong*

Le type "*xs:unsignedLong*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 64 bits, soit les valeurs allant de 0 à +18446744073709551615 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedLong*" est obtenu par restriction à partir de "*xs:nonNegativeInteger*".

#### > *xs:unsignedInt*

Le type "*xs:unsignedInt*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 32 bits, soit les valeurs allant de 0 à +4294967295 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedInt*" est obtenu par restriction à partir de "*xs:unsignedLong*".

#### > *xs:unsignedShort*

Le type "*xs:unsignedShort*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 16 bits, soit les valeurs allant de 0 à +65535 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedShort*" est obtenu par restriction à partir de "*xs:unsignedInt*".

#### > *xs:unsignedByte*

Le type "*xs:unsignedByte*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 8 bits, soit les valeurs allant de 0 à +255 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedByte*" est obtenu par restriction à partir de "*xs:unsignedShort*".



### 5.3.7 Nombres à virgule flottante

#### > xs:float

Le type `"xs:float"` s'adresse aux nombres à virgule flottante. La représentation utilisée dans l'espace des valeurs sera conforme au standard **IEEE 754** simple précision.

La représentation standard **IEEE 754** en simple précision met tout d'abord le nombre à représenter sous la forme donnée ci-dessous, en réservant le bit de poids fort pour le signe, suivi par un exposant entier sur 8 bits et 23 bits pour la mantisse :

$$x = \pm m \cdot 2^e$$

Vue la représentation employée, l'exposant se verra compris dans l'intervalle  $[-126, +127]$ , tandis que la mantisse prendra des valeurs comprises entre 1 et  $2 \cdot 2^{-23}$ . De fait, l'intervalle utile, exprimé en base 10, va approximativement de  $10^{-38}$  à  $10^{38}$  pour les nombres positifs ou négatifs.

**N.B.** Il est évident qu'il n'est pas possible de représenter de manière exacte n'importe quel nombre réel sous cette forme. La valeur retenue dans l'espace des valeurs pourra donc être uniquement une approximation de la représentation lexicale (*valeur la plus proche*).

Le standard **IEEE 754** prévoit des cas particuliers :


- Il y a de fait deux représentations possibles pour le zéro : `"0"` ou `"+0"` pour le zéro positif, et `"-0"` pour le zéro négatif.
- Il existe de plus une représentation particulière pour un nombre plus grand que le plus grand des nombres que l'on peut représenter, c'est à dire *"plus l'infini"* noté `"INF"`. Il y a de même une représentation pour *"moins l'infini"* notée `"-INF"`.
- Enfin, lorsqu'un calcul renvoie une valeur non représentable (*comme 0 fois l'infini*) il existe un code particulier, noté `"NaN"` qui signifie *"Not a Number"*.

L'espace lexical d'un nombre du type `"xs:float"` admet donc les représentations suivantes :

- `"2.1"` : un nombre décimal (*point obligatoire*) précédé d'un signe optionnel,
- `"314.e-2"` : un nombre décimal précédé d'un signe optionnel suivi par la lettre `"e"` ou `"E"`, suivie par un exposant entier éventuellement précédé par un signe optionnel (*aucun espace permis*),
- l'une des valeurs `"INF"`, `"-INF"`, ou `"NaN"` (*attention, +INF est interdit*).

```
<xs:attribute name="valeur" type="xs:float"/>
```

```
<info desc="distance terre-soleil">
  <minimum valeur="147.1e6" unité="km">
  <maximum valeur="152.1e6" unité="km">
  <moyenne valeur="149.6e6" unité="km">
</info>
```

 Page de cours sur le codage IEEE 754

[\[http://dmolinarius.github.io/demofiles/microp/numeration/flp\\_ieee.html\]](http://dmolinarius.github.io/demofiles/microp/numeration/flp_ieee.html)

#### > xs:double

Le type `"xs:double"` concerne encore les nombres à virgule flottante. La représentation utilisée dans l'espace des valeurs sera conforme au standard **IEEE 754** double précision.

La seule différence se situe dans la taille de la zone mémoire requise pour stocker la valeur : 64 bits au lieu de 32 bits pour `"xs:float"`. Du coup la taille de l'exposant passe à 11 bits tandis que l'espace réservé à la mantisse comporte 52 bits.

L'exposant se verra par conséquent compris dans l'intervalle  $[-1022, +1023]$ , tandis que la mantisse prendra des valeurs comprises entre 1 et  $2 \cdot 2^{-52}$ . L'intervalle utile, exprimé en base 10, va donc approximativement de  $10^{-308}$  à  $10^{308}$  pour les nombres positifs ou négatifs.

## 5.4 Création de types simples

### 5.4.1 Principe des dérivations

**XML Schema** permet de créer des types personnalisés à partir de types existants (*prédéfinis ou personnalisés*). Le mécanisme permettant de fabriquer ses propres types de données s'appelle la **dérivation**.

Il existe trois méthodes de dérivation permettant de créer de nouveaux types simples : la **restriction**, la **liste** et l'**union**.

La **restriction** consiste à augmenter les contraintes sur les données autorisées, ce qui conduit à diminuer la taille de l'espace lexical. Les facettes qui s'appliquent au type de données originel s'appliquent également au nouveau type restreint.

```
<xs:attribute name="promo" type="T_PROMO"/>
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1999"/>
    <xs:maxInclusive value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple1.html)

La **liste** permet de créer des listes de données appartenant toutes au même type. Les listes possèdent leurs propres facettes et perdent celles des membres qui les composent.

```
<xs:simpleType name="idCours">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:pattern value="[A-Z] [A-Z] ([1-9] | 10)"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
<xs:attribute name="cours" type="idCours"/>
```

```
<étudiant id="e34" cours="TI1 IF2 TI4 PH7 TI8 IF9"/>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple2.html)

L'**union** revient à fusionner les espaces lexicaux de divers types simples. La sémantique attachée à chacun des types d'origine est perdue. Le type résultant ne garde que les facettes spécifiques aux unions, à savoir "**xs:pattern**" et "**xs:enumeration**".

```
<xs:simpleType name="profId">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:pattern value="inconnu"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:IDREF"/>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

```
<cours id="IF3" profId="inconnu">
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple3.html>]

### 5.4.2 Dérivation par restriction

La dérivation par **restriction** consiste à créer un nouveau type en renforçant les contraintes sur les données d'un type existant (*appelé type de base*). Ceci revient donc à ce que l'espace lexical du type créé soit un sous-ensemble de celui du type de base.

La restriction s'effectue en agissant sur certaines caractéristiques spécifiques du type de base appelées **facettes**. La grande majorité des facettes agissent sur l'espace des valeurs. Une facette de même nom peut avoir des effets différents selon le type de base auquel on applique la restriction.

L'exemple ci-dessous montre une restriction du type de base "*xs:unsignedShort*" agissant sur les facettes "*xs:minInclusive*" et "*xs:maxInclusive*" pour limiter l'intervalle des valeurs possibles :

```
<xs:simpleType name="type-promo">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1999"/>
    <xs:maxInclusive value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/restriction-exemple1.html>]

Chaque type prédéfini possède une liste de facettes qui s'appliquent à lui. Cette liste est fixe, non modifiable et non extensible.

Quelle que soit la facette utilisée, la sémantique attachée au type de base est conservée, ce qui concerne en particulier les règles de passage de l'espace lexical à l'espace des valeurs, mais aussi la liste des facettes qui s'appliqueront au type créé.

```
<xs:simpleType name="pseudo-nombre">
  <xs:restriction base="xs:NMTOKEN">
    <xs:pattern value="[0-9] +"/>
  </xs:restriction>
</xs:simpleType>
```

L'exemple ci-dessus montre à titre d'illustration une restriction à partir du type "*xs:NMTOKEN*" qui résulte en un type n'acceptant que les chiffres. Pourtant, l'espace des valeurs de ce type nouveau sera tout à fait différent (*il s'agira de chaînes de caractères*) de celui de "*xs:integer*" (*des nombres*).

### 5.4.3 Facette xs:pattern

#### > xs:pattern

La facette "*xs:pattern*" est particulière, puisque c'est la seule qui exprime des contraintes sur l'espace lexical pour limiter l'espace des valeurs. Toutes les autres facettes utiles agissent sur l'espace des valeurs.

Cette facette s'applique à tous les types primitifs, ainsi que les types prédéfinis dérivés par restriction (*ce qui exclut les listes xs:NMTOKENS et xs:IDREFS*).

"*xs:pattern*" s'appuie sur une **expression régulière** pour indiquer quels sont les motifs que doivent respecter les formes lexicales du type résultant.

Les expressions régulières sont dérivées de celles du langage **Perl** (*Practical Extraction and Reporting Language*). Leur syntaxe est particulièrement puissante, mais aussi complexe et ardue à appréhender. C'est la raison pour laquelle nous nous contenterons ici de donner quelques exemples.

Liste de valeurs :

L'usage le plus simple de la facette "**xs:pattern**" consiste à créer des listes de possibilités :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999"/>
    <xs:pattern value="2000"/>
    <xs:pattern value="2001"/>
    <xs:pattern value="2002"/>
    <xs:pattern value="2003"/>
    <xs:pattern value="2004"/>
    <xs:pattern value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple1.html>]

L'intérêt par rapport à l'usage d'une restriction par énumération réside dans le fait que la facette "**xs:pattern**" agit sur l'**espace lexical**, alors que la facette "**xs:enumeration**" agit sur l'**espace des valeurs**. Cela signifie qu'une expression comme "**02000**" ne sera pas validée dans le cas présent, alors qu'elle le serait dans le cas d'une énumération.

**Alternatives :**

Le même effet que ci-dessus aurait pu être obtenu en utilisant l'opérateur "**|**" qui permet de signifier des alternatives :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999|2000|2001|2002|2003|2004|2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple2.html>]

**Classes de caractères :**

Il est possible de spécifier des intervalles de caractères admis à l'aide des opérateurs "**[**" et "**]**". L'exemple précédent devient :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999|200[0-5]"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple3.html>]

**Grouperment de sous-motifs :**

Les sous-motifs peuvent être groupés à l'aide de parenthèses. Ainsi, L'exemple ci-dessous permet de représenter un sigle de la forme "**TI-8**", constitué de deux lettres majuscules, suivies par un tiret obligatoire, suivi par un nombre de **1** à **10** :

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z][A-Z] - ([1-9]|10)"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple4.html>]

On peut remarquer que vu le choix du type de base, les sigles appartenant au type ci-dessus garderont la sémantique d'un identifiant unique...

### Opérateurs de répétition :

Les caractères individuels, classes, ou groupes (*délimités par des parenthèses*) peuvent être post-fixés par des opérateurs de répétition "?" (zéro ou un), "+" (un ou plus), "\*" (zéro ou plus).

Le type défini ci-dessous accepte les sigles formés de 1 ou 2 caractères majuscules, suivi par un tiret et un nombre non nul d'un chiffre au moins, ne commençant pas par le chiffre 0.

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z] [A-Z] ?-[1-9] [0-9] *"/>
  </xs:restriction>
</xs:simpleType>
```

#### Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple5.html>]

### Opérateur de répétition généralisé :

Dans le cas général, la notation "{n,m}" indique la possibilité d'accepter de n à m occurrences. La définition ci-dessus pourrait alors s'écrire :

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z] {1,2} - [1-9] [0-9] {0,}"/>
  </xs:restriction>
</xs:simpleType>
```

#### Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple6.html>]

## 5.4.4 Facette xs:enumeration

### > xs:enumeration

La facette "xs:enumeration" est la seule à s'appliquer à la fois aux nombres, aux listes, et aux chaînes de caractères. Elle permet tout simplement de lister l'ensemble des valeurs autorisées :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:enumeration value="1999"/>
    <xs:enumeration value="2000"/>
    <xs:enumeration value="2001"/>
    <xs:enumeration value="2002"/>
    <xs:enumeration value="2003"/>
    <xs:enumeration value="2004"/>
    <xs:enumeration value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

#### Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple1.html>]

Il est important de se rendre compte que, même si l'expression qui apparaît comme valeur de l'attribut "value" le fait forcément par l'intermédiaire d'une valeur lexicale (*on est obligé de l'écrire*), la sélection se fait sur l'espace des valeurs. Dans l'exemple ci-dessus, la valeur 02005 est ainsi validée.

Cette remarque est particulièrement importante dans le cas des nombres à virgule flottante qui acceptent un grand nombre d'orthographes (*i.e. de valeurs lexicales*) différentes pour une même valeur (*dans l'espace des valeurs*).

Ceci s'applique aussi dans le cas de chaînes de caractères compactées du type *"xs.token"*. Dans ce cas, *"xs.enumeration"* permet de spécifier les items autorisés, aux espaces consécutifs près :

```
<xs:simpleType name="T_OPTION">
  <xs:restriction base="xs:token">
    <xs:enumeration value="Informatique"/>
    <xs:enumeration value="Technologies de l'Information et de la
Communication"/>
  </xs:restriction>
</xs:simpleType>
```

Dans ce cas, des chaînes comme " *Informatique* " ou "*Technologies*<TAB>*de l'Information*<CR><TAB>*et de la Communication*" sont validées.

Constater de visu :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple2.html>]

Il faut se méfier par contre du cas ci-dessous, où justement, les chaînes conformes aux espaces près ne seraient pas validées :

```
<xs:simpleType name="T_OPTION">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Informatique"/>
    <xs:enumeration value="Technologies de l'Information et de la
Communication"/>
  </xs:restriction>
</xs:simpleType>
```

Vérifier ce fait :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple3.html>]

**N.B.** La facette *"xs.enumeration"* ne s'applique pas au type *"xs:boolean"*.

#### 5.4.5 Facettes xs:length et al.

##### > xs:length

**Chaînes de caractères :**

La facette *"xs:length"* s'applique au type *"xs:string"* et aux types qui en sont dérivés par restriction. Elle permet de spécifier le nombre de caractères requis dans l'**espace des valeurs**.

```
<xs:simpleType name="T_SIGLE_OPT">
  <xs:restriction base="xs:NCName">
    <xs:pattern value="[A-Z]*/>
    <xs:length value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Dans l'exemple ci-dessus, les valeurs autorisées sont toutes les combinaisons de deux caractères majuscules.

Vérifier :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple1.html>]

**N.B.** Cette facette permet très simplement la création d'un type d'élément vide, puisqu'il suffit de la forcer à 0 :

```
<xs:simpleType name="T_VIDE">
  <xs:restriction base="xs:string">
    <xs:length value="0"/>
  </xs:restriction>
</xs:simpleType>
```

### Flux binaires :

"*xs:length*" s'applique également aux types "*xs:hexBinary*" et "*xs:base64Binary*" pour lesquels elle spécifie le nombre d'octets :

```
<xs:element name="Authorization" type="T_HTTP_PASSWORD"/>
<xs:simpleType name="T_HTTP_PASSWORD">
  <xs:restriction base="xs:base64Binary">
    <xs:length value="13"/>
  </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, la longueur mentionnée correspond à l'**espace des valeurs** et valide effectivement la chaîne passée ci-dessous qui fait pourtant bien plus de **13** caractères, mais correspond à la valeur "*be-http:cool!*" (qui elle comporte bien 13 caractères).

```
<Authorization>YmUtaHR0cDpjb29sIQ==</Authorization>
```

#### Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple2.html>

### Listes :

Enfin, "*xs:length*" s'applique à tous les types dérivés par liste (*prédéfinis ou non*). Cette facette sert dans ce cas à indiquer le nombre exact d'items de liste autorisés.

```
<xs:attribute name="élèves" type="T_MOLECULE_LIST"/>
<xs:simpleType name="T_MOLECULE_LIST">
  <xs:restriction base="xs:IDREFS">
    <xs:length value="6"/>
  </xs:restriction>
</xs:simpleType>
```

Le schéma ci-dessus permet de valider l'extrait suivant, car la liste d'élèves d'une molécule comporte effectivement **6** items :

```
<molécule id="M33" élèves="E262 E133 E486 E532 E122 E44"/>
```

#### Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple3.html>

**N.B.** En toute rigueur, "*xs:length*" s'applique également aux types "*xs:QName*" et "*xs:anyURI*". Toutefois, les spécifications sont plus que vagues sur la façon de compter des caractères dans les espaces de valeurs très particuliers de ces deux types de données.

#### > *xs:maxLength*

Cette facette est similaire à "*xs:length*", sauf qu'elle indique le nombre **maximal** de caractères et non pas le nombre **exact**. Elle s'applique aux mêmes types, de la même façon, et appelle les mêmes remarques que "*xs:length*".

#### > *xs:minLength*

De même, "*xs:minLength*" est en tous points comparable à "*xs:maxLength*", mis à part qu'elle permet de spécifier le nombre **minimal** et non pas le nombre **maximal** de caractères.

A titre d'exemple, la molécule de l'exemple suivant peut contenir soit **5** soit **6** élèves :

```
<xs:attribute name="élèves" type="T_MOLECULE_LIST"/>
<xs:simpleType name="T_MOLECULE_LIST">
  <xs:restriction base="xs:IDREFS">
    <xs:minLength value="5"/>
    <xs:maxLength value="6"/>
  </xs:restriction>
</xs:simpleType>
```



Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple4.html>]

#### 5.4.6 Facettes min et max

Les facettes décrites dans cette section s'appliquent aux types dont l'espace des valeurs est ordonné. C'est en particulier le cas des nombres, entiers, décimaux ou à virgule flottante, et de leurs types dérivés par restriction, mais pas des chaînes de caractères ni des types dérivés.

##### > **xs:maxInclusive**

"**xs:maxInclusive**" permet de fixer la borne supérieure de l'espace des valeurs. La valeur fournie fait partie de l'intervalle autorisé.

```
<xs:simpleType name="T_ALTITUDE">
  <xs:restriction base="xs:float">
    <xs:maxInclusive value="8848."/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple1.html>]

##### > **xs:maxExclusive**

"**xs:maxExclusive**" permet de fixer la borne supérieure de l'espace des valeurs. La valeur fournie ne fait pas partie de l'intervalle autorisé. Cette propriété est plus particulièrement utile pour les nombres décimaux ou à virgule flottante.

```
<xs:simpleType name="T_DECIMAL_NEGATIF">
  <xs:restriction base="xs:decimal">
    <xs:maxExclusive value="0."/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple2.html>]

##### > **xs:minInclusive**

"**xs:minInclusive**" permet de fixer la borne inférieure de l'espace des valeurs. La valeur fournie fait partie de l'intervalle autorisé.

```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minInclusive value="-273.15"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple3.html>]

##### > **xs:minExclusive**

"**xs:minExclusive**" permet de fixer la borne inférieure de l'espace des valeurs. La valeur fournie ne fait pas partie de l'intervalle autorisé. Cette propriété est plus particulièrement utile pour les nombres décimaux ou à virgule flottante.

```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minExclusive value="-273.15"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple4.html>]



### 5.4.7 Facettes de digits

#### > xs:totalDigits

La facette "*xs:totalDigits*" s'applique au type "*xs:decimal*" et à ses types dérivés dont font partie tous les types entiers. Elle sert à fixer le nombre **maximum** de chiffres significatifs (*caractères 0 à 9*), ce qui inclut les chiffres après la virgule dans le cas des décimaux.

L'exemple ci-dessous définit un type permettant d'exprimer des années allant de l'an 0 à l'année 9999 :

```
<xs:simpleType name="T_ANNEE">
  <xs:restriction base="xs:unsignedShort">
    <xs:totalDigits value="4"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/digits-exemple1.html>]

On rappelle que les facettes agissent sur l'espace des valeurs ce qui explique qu'une forme lexicale comme "*02005*" soit validée pour le type de l'exemple ci-dessus.

#### > xs:fractionDigits

Cette facette ne s'applique qu'au seul type prédéfini "*xs:decimal*" (et aux éventuels types non prédéfinis dérivés par restriction). Elle fixe le nombre maximal de chiffres significatifs après la virgule.

```
<xs:simpleType name="T_PRIX">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/digits-exemple2.html>]

Là encore, la facette agit sur l'espace des valeurs ce qui signifie que l'expression "*1.4200*" sera validée, contrairement à "*1.123*" par exemple.

### 5.4.8 Restrictions multiples

Il est possible d'obtenir un nouveau type dérivant par restriction d'un type lui-même obtenu à l'aide d'une dérivation par restriction.

Lorsque les restrictions portent sur des facettes différentes, les conditions s'ajoutent pour créer un type dont l'espace des valeurs est à chaque fois plus petit, puisqu'il est inclus (*en général strictement*) dans l'espace de valeurs du type de base.

Lorsqu'une restriction porte par contre sur une facette déjà restreinte par le type de base, alors certaines conditions sont à respecter :

- la facette "*xs:length*", une fois fixée, ne peut plus être modifiée par un type dérivé,
- la plupart des facettes peuvent être rédéfinies à condition d'être plus restrictives que pour le type de base (*facettes d'énumération, maximums, minimums, et contrôle des digits*),
- les applications consécutives de la facette "*xs:pattern*" imposent un respect simultané de tous les motifs. L'espace lexical résultant est donc l'intersection des espaces correspondant à chacun des motifs.

Lors de la création d'un type il est possible d'interdire toute modification ultérieure (*cf. types dérivés*) d'une facette à l'aide de l'attribut "*fixed*". Cet attribut est disponible pour l'ensemble des facettes à l'exception de "*xs:enumeration*" et "*xs:pattern*" :

```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minExclusive value="-273.15" fixed="true"/>
  </xs:restriction>
</xs:simpleType>
```

Enfin, comme l'on montré quelques-uns des exemples déjà vus, il est également possible d'appliquer des restrictions qui agissent sur plusieurs facettes à la fois :

```
<xs:simpleType name="T_PRIX">
  <xs:restriction base="xs:decimal">
    <xs:pattern value="[0-9]*\.[0-9]{2}"/>
    <xs:maxInclusive value="10.0"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/restmult-exemple2.html>]

Le type ci-dessus pourrait s'appliquer aux prix d'un magasin dont tous les articles ont un prix inférieur ou égal à **10.0**. Les représentations lexicales autorisées comporteront obligatoirement deux chiffres après la virgule. Les zéros non significatifs au-delà du deuxième chiffre après la virgule, comme pour la chaîne **"9.540"**, sont interdits. Les zéros non significatifs en tête de nombre, comme pour **"04.50"** sont autorisés. Les signes (*et par conséquent, les valeurs négatives*) sont interdits.

#### 5.4.9 Dérivation par liste

La dérivation par liste permet d'obtenir un type **"liste"** dont tous les items sont du même type. **"xs:IDREFS"** et **"xs:NMTOKENS"** sont deux exemples de listes prédéfinies, mais il est possible de créer des listes à partir de n'importe quel type simple, prédéfini ou non :

```
<xs:attribute name="coords" type="T_RECT_COORDS">
<xs:simpleType name="T_RECT_COORDS">
  <xs:list itemType="xs:decimal"/>
</xs:simpleType>
```

```
<rectangle coords="10.0 10.0 980.0 150.0"/>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple1.html>]

La définition ci-dessus aurait également pu être effectuée en embarquant une création de type simple comme contenu de l'élément **"xs:liste"** plutôt que de référencer un type global à l'aide de l'attribut **"itemType"** :

```
<xs:simpleType name="T_RECT_COORDS">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:decimal">
        <xs:pattern value="[0-9]{1,3}\.[0-9]?"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple2.html>]

#### > Facettes

Les types **"liste"** perdent les facettes du type de leurs items et possèdent leurs facettes propres **"xs:length"**, **"xs:minLength"**, **"xs:maxLength"** et **"xs:enumeration"**.

Pour appliquer une restriction sur une facette d'un type liste créé, ce type doit forcément être global. Le type défini ci-dessous comporte une liste comportant exactement 4 coordonnées :

```
<xs:simpleType name="T_RECT_COORDS">
  <xs:restriction base="T_COORDS">
    <xs:length value="4"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="T_COORDS">
  <xs:list itemType="xs:decimal"/>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple3.html>]

**N.B.** La raison imposant que le type des éléments de la liste soit global est qu'un type simple n'accepte qu'une seule méthode de dérivation à la fois (*restriction, liste, ou union*).

### > Limitations

Les listes obtenues de cette manière sont très limitées. En effet, les items de liste sont obligatoirement séparés par un espace (*après compactage*), et il n'est pas possible de spécifier un autre caractère de séparation. Il n'est pas possible non plus de créer des listes dont les éléments n'appartiennent pas au même type simple, ou de créer des listes d'éléments complexes, ni même des listes de listes (*nommément interdit par les spécifications*)...

Lorsque les valeurs du type servant d'item de liste sont susceptibles de contenir des espaces (*comme c'est le cas de xs:string ou xs:token*), alors ces blancs seront pris en compte comme séparateurs de liste, ce qui fait que la facette "*xs:length*" compterait alors des "*mots*" (*en fait, des unités lexicales ne comprenant pas d'espace*)...

```
<xs:element name="summary" type="T_SUMMARY">
  <xs:simpleType name="T_SUMMARY">
    <xs:restriction base="T_WORDS">
      <xs:maxLength value="1000"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="T_WORDS">
    <xs:list itemType="xs:token"/>
  </xs:simpleType>
```

```
<summary>Ceci est un résumé limité à 1000 mots...</summary>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple4.html>]

### > Remarque

En terme de conception d'applications **XML**, il n'est pas très pertinent d'abuser des types "*liste*", qu'ils servent à recevoir des valeurs d'attributs ou d'éléments. En effet, les mécanismes classiques (SAX, DOM, XPath, ...) ne permettent pas facilement de récupérer les valeurs individuelles des items d'une liste, contrairement à ce qui serait possible avec une liste d'éléments :

```
<molécule id="M33" élèves="E262 E133 E486 E532 E122 E44"/>
```

La solution ci-dessous serait donc si possible préférable :

```
<molécule>
  <élève ref="E262"/>    <élève ref="E133"/>
  <élève ref="E486"/>    <élève ref="E532"/>
  <élève ref="E122"/>    <élève ref="E44"/>
</molécule>
```

### 5.4.10 Dérivation par union

L'union est une méthode de dérivation qui permet de fusionner les espaces de valeurs de types distincts.

L'exemple ci-dessous permet de rajouter au type "*xs:decimal*" la chaîne de caractère "*undefined*" :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:decimal"/>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:enumeration value="undefined"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Dans le cas ci-dessus, il est possible de raccourcir un peu le code par l'intermédiaire de l'attribut "*memberTypes*" qui permet de spécifier une liste de types membres de l'union :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union memberTypes="xs:decimal">
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:enumeration value="undefined"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Ou encore :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union memberTypes="xs:decimal T_UNDEF"/>
</xs:simpleType>

<xs:simpleType name="T_UNDEF">
  <xs:restriction base="xs:NCName">
    <xs:enumeration value="undefined"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/union-exemple3.html>

#### > Facettes

Les seules facettes qui peuvent s'appliquer aux types dérivés par union sont : "*xs:pattern*" et "*xs:enumeration*".

## 5.5 Création de types complexes

### 5.5.1 Introduction

Les types simples tels que tous ceux vus jusqu'ici décrivent tous le contenu d'un noeud texte (*élément ou attribut*). Ils n'ont aucun rapport avec un quelconque balisage, et peuvent mener une existence indépendante de **XML Schema** voire même de **XML**.

C'est la raison pour laquelle les spécifications de **XML Schema** comportent deux parties indépendantes, dont l'une est justement consacrée aux types de données prédéfinis.

*A contrario*, les types complexes décrivent la structure du balisage. Ils sont donc généralement plus spécifiques d'une application et moins réutilisables dans un cadre différent.

Toutefois, et comme pour les types simples, les types complexes peuvent être nommés et définis à l'échelon global, ou alors être anonymes et définis localement en fonction du besoin.

### Exemple de définition globale

```
<xs:element name="étudiant" type="T_ETUDIANT"/>
<xs:element name="prénom" type="xs:token"/>
<xs:element name="nom" type="xs:token"/>

<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

### Exemple de définition locale

```
<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Rappelons également qu'on appelle *"complexe"*, le type de tout élément qui ne soit pas à contenu purement **textuel**. Ceci inclut les éléments à contenu textuel munis d'attributs qui seront dits *"éléments complexes à contenu simple"*, tous les autres étant dits *"à contenu complexe"* :

● élément de type simple :

```
<nom>Deubaze</nom>
```

● élément de type complexe à contenu simple :

```
<personne civilité="M.">Raymond Deubaze</personne>
```

● éléments de type complexe à contenu complexe :

```
<personne>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom> </
personne>
```

```
<personne civilité="M.">
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom> </
personne>
```

```
<personne>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
  Un très bon élève... </
personne>
```

### 5.5.2 Création de type complexe à contenu simple

Un type complexe à contenu simple, obtenu en ajoutant un attribut à un type simple, est dit **créé par extension** :

```
<personne civilité="M.">Raymond Deubaze</personne>
```

```
<xs:complexType name="T_PERSONNE">
  <xs:simpleContent>
    <xs:extension base="T_NOM_PERSONNE">
      <xs:attribute name="civilité" type="T_CIVILITE"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="T_NOM_PERSONNE">
  <xs:restriction base="xs:token">
    <xs:pattern value="[A-Z][a-z]+ [A-Z][a-z]+">
  </xs:restriction>
</xs:simpleType>
```

L'attribut **"base"** est obligatoire. La définition du type simple ne peut pas se faire localement, comme contenu de l'élément **"xs:extension"**.

### 5.5.3 Dérivation de type complexe à contenu simple

Un type complexe à contenu simple obtenu en ajoutant un attribut à un autre type complexe à contenu simple est dit **dérivé** par extension :

Par rapport au cas précédent la nuance est subtile, d'autant plus que la syntaxe est strictement identique :

```
<xs:complexType name="T_PERSONNE_NATION">
  <xs:simpleContent>
    <xs:extension base="T_PERSONNE">
      <xs:attribute name="nationalité" type="T_NATION"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

```
<personne civilité="M." nationalité="F">Raymond Deubaze</personne>
```

La seule différence par rapport au cas précédent tient dans le fait que **"T\_PERSONNE"** est un type complexe à contenu simple au lieu d'un type simple...

### 5.5.4 Restriction de type complexe à contenu simple

La dérivation par restriction d'un type complexe à contenu simple apporte une nouveauté par rapport à la restriction d'un type simple : elle agit non seulement sur le contenu textuel mais également sur les attributs.

La syntaxe permettant de restreindre le contenu textuel est la même que pour les éléments simples. Un attribut peut être restreint en modifiant son type pour un type restreint du précédent, ou tout simplement interdit.

#### Exemple : type de départ

```
<xs:complexType name="T_PERSONNE">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="nationalité" type="xs:NMTOKEN"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

### Restriction du contenu :

```
<xs:complexType name="T_NOM_PRENOM">
  <xs:simpleContent>
    <xs:restriction base="T_PERSONNE">
      <xs:pattern value="[A-Z][a-z]+ [A-Z][a-z]+" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

### Restriction du type de l'attribut :

```
<xs:complexType name="T_FRANCAIS">
  <xs:simpleContent>
    <xs:restriction base="T_NOM_PRENOM">
      <xs:attribute name="nationalité" type="T_NAT_F" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="T_NAT_F">
  <xs:restriction base="xs:NMTOKEN">
    <xs:pattern value="F" />
  </xs:restriction>
</xs:simpleType>
```

**N.B.** la restriction de l'attribut aurait également pu être effectuée avec un type local :

```
<xs:complexType name="T_FRANCAIS">
  <xs:simpleContent>
    <xs:restriction base="T_NOM_PRENOM">
      <xs:attribute name="nationalité">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:pattern value="F" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

### Suppression de l'attribut :

La suppression totale de l'attribut s'effectue à l'aide de la valeur *"prohibited"* de l'attribut *"use"*, comme dans l'exemple ci-dessous :

```
<xs:complexType name="T_APATRIDE">
  <xs:simpleContent>
    <xs:restriction base="T_FRANCAIS">
      <xs:attribute name="nationalité" type="xs:NMTOKEN" use="prohibited" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

### Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/simpleContent-exemple3.html>

## 5.5.5 Création de type complexe à contenu complexe

Un **type complexe à contenu complexe** définit la liste des sous-éléments autorisés, l'ordre dans lequel ils peuvent ou doivent apparaître et le nombre d'occurrences autorisées.

### > xs:sequence

La façon la plus simple de créer un type complexe consiste à lister les sous-éléments dans l'ordre dans lequel ils sont autorisés à apparaître, à l'aide d'un élément *"xs:sequence"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

**N.B.** On rappelle que les éléments listés par *"xs:sequence"* peuvent également être nommés (via l'attribut *name*) et définis localement au lieu d'être définis globalement puis référencés (via l'attribut *ref*) :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element name="prénom" type="T_NOM"/>
    <xs:element name="nom" type="T_NOM"/>
  </xs:sequence>
</xs:complexType>
```

Et la logique peut même être poussée encore plus loin, en définissant localement les types des éléments. (cf. ci-dessus *nom* et *prénom*)...

### > Contrôle du nombre d'occurrences

Le nombre d'occurrences autorisées pour chacun des éléments peut être indiqué à l'aide des attributs *"minOccurs"* et *"maxOccurs"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom" maxOccurs="3"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Par défaut ces attributs valent **1**, ce qui rend obligatoire une unique occurrence de l'élément concerné.

Pour rendre un élément optionnel, il suffit de mettre la valeur *"minOccurs"* à **0**. Pour ne pas limiter le nombre d'occurrences, il existe la valeur particulière *"unbounded"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom" maxOccurs="unbounded"/>
    <xs:element ref="prénom-usuel" minOccurs="0"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

### > xs:choice

Comme avec les **DTD**, il est possible d'offrir des alternatives concernant les listes possibles de sous-éléments. Cette aspect est contrôlé par l'élément *"xs:choice"*, qui permet de proposer une liste d'éléments mutuellement exclusifs :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="prénom"/>
      <xs:element ref="prénom-usuel"/>
    </xs:choice>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```



Cet exemple impose l'usage d'un prénom ou (*exclusif*) d'un prénom usuel, suivi par un nom.

**N.B.** On remarque ici que l'élément `"xs:choice"` a pris la place d'un élément `"xs:element"` dans la liste spécifiée par `"xs:sequence"`. En effet, les éléments, les séquences, les choix, constituent des sortes de briques, ou **particules** (*particles*) qui peuvent apparaître chacune en lieu et place d'un élément comme membre d'une séquence ou d'un choix.

On peut mettre en pratique cette propriété pour imposer par exemple un prénom unique ou alors un prénom usuel éventuellement suivi par un ou plusieurs prénoms :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="prénom"/>
      <xs:sequence>
        <xs:element ref="prénom-usuel"/>
        <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:choice>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple3.html>]

### > xs:group

En raisonnant sur notre exemple, le choix entre un prénom unique ou un prénom usuel suivi par une liste de prénoms, peut être une brique de base que nous aurions envie de réutiliser par ailleurs.

Or il ne s'agit pas là d'un élément que nous pourrions rendre global, ni même d'un type. C'est là qu'intervient l'élément `"xs:group"` qui consitue une autre brique ou particule réutilisable :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>

<xs:group name="G_PRENOMS">
  <xs:choice>
    <xs:element ref="prénom"/>
    <xs:sequence>
      <xs:element ref="prénom-usuel"/>
      <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:choice>
</xs:group>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple4.html>]

### > xs:attribute

La déclaration des attributs d'un type complexe à contenu complexe s'effectue à l'aide d'éléments `"xs:attribute"` placés **après** la brique (*ou particule*) constituant le type :

```
<xs:complexType name="T_PERSONNE">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>
```

De la même manière qu'il a pu être utile d'extraire des groupes structurels d'éléments sous forme de briques afin de pouvoir les réutiliser, il est intéressant, dès que la complexité d'une application devient moyenne, de pouvoir grouper des séries d'attributs couramment utilisés ensemble :

```
<xs:complexType name="T_PERSONNE">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
  <xs:attributeGroup ref="A_COURANTS"/>
</xs:complexType>

<xs:attributeGroup name="A_COURANTS">
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="nationalité" type="xs:NCName"/>
</xs:attributeGroup>
```

La plupart du temps les attributs sont optionnels. Toutefois, il est possible de rendre un attribut obligatoire à l'aide de l'attribut **"use"** prenant la valeur **"required"** :

```
<xs:attributeGroup name="A_COURANTS">
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="nationalité" type="xs:NCName" use="required"/>
</xs:attributeGroup>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple5.html>]

### 5.5.6 Dérivation de type complexe à contenu complexe

#### > Dérivation par extension

La dérivation par extension d'un type complexe à contenu complexe permet d'ajouter des éléments ou des attributs au type de base (*noter l'élément **xs:complexContent** caractéristique d'une dérivation*) :

```
<xs:complexType name="T_ETUDIANT">
  <xs:complexContent>
    <xs:extension base="T_PERSONNE">
      <xs:sequence>
        <xs:element name="email" type="xs:token"/>
      </xs:sequence>
      <xs:attribute name="promo" type="xs:integer" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple6.html>]

Malheureusement, ce mécanisme reste très limité. En effet, il n'est possible de compléter le contenu du type de base qu'en rajoutant une brique (*ou particule*) **après le contenu existant**.

Tout se passe comme si on avait un nouveau type construit de la manière suivante :

```
<xs:complexType>
  <xs:sequence>
    <!-- contenu de l'élément de base -->
    <!-- contenu ajouté par extension -->
  </xs:sequence>
</xs:complexType>
```

Il n'est donc pas possible par extension de rajouter un élément à une alternative **"xs:choice"** ou de choisir un autre point d'insertion du contenu venant compléter le type de base.

### 5.5.7 Type complexe à contenu mixte

**XML Schema** traite les éléments complexes à contenu mixte comme un cas particulier des éléments complexes à contenu complexe.

Supposons que l'on veuille pouvoir ajouter des commentaires libres, hors éléments, pour qualifier les étudiants de notre exemple :

```
<étudiant>
  Un peu de pipotage ne lui ferait pas de mal !
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
</étudiant>
```

Il suffirait tout simplement d'ajouter l'attribut *"mixed"* avec la valeur *"true"* à la déclaration de type :

```
<xs:complexType name="T_ETUDIANT" mixed="true">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Cette façon d'aborder le problème permet de conserver tous les mécanismes vus pour les éléments complexes à contenu complexe, mais interdit tout contrôle quant à la position du texte qui peut apparaître n'importe où entre les éléments :

```
<étudiant>
  <prénom>Ginette</prénom>
  <nom>Ringard</nom>
  Devrait se mettre à 1'heure d'Internet...
</étudiant>

<étudiant>
  <prénom>Jean</prénom>
  Se décourage facilement.
  <nom>Aymard</nom>
</étudiant>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/mixedContent-exemple1.html>

### 5.5.8 Type complexe à contenu vide

Selon **XML Schema**, les éléments vides peuvent être indifféremment considérés comme des éléments à contenu textuel de longueur nulle, ou comme des éléments à contenu élémentaire sans enfants.

Conformément à la seconde hypothèse, un type vide se définit donc de la manière suivante :

```
<xs:complexType name="T_VIDE">
  <xs:sequence/>
</xs:complexType>
```

Un type vide portant des attributs n'est guère plus compliqué à définir :

```
<xs:complexType name="T_PROMO">
  <xs:sequence/>
  <xs:attribute name="année" type="xs:integer"/>
</xs:complexType>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/emptyContent-exemple1.html>

## 5.6 Validation

Il existe de nombreux outils compatibles avec **XML Schema** (*éditeurs, validateurs, ...*). Une bonne idée serait sans doute (*car un peu datée*) de consulter la page de référence sur le site du **W3C** qui indique notamment un certain nombre d'analyseurs validants open source et un module de validation pour divers langages comme **C/C++**, **Java**, **Perl** ou **Python** :

 <http://www.w3.org/XML/Schema#Tools>

[\[http://www.w3.org/XML/Schema#Tools\]](http://www.w3.org/XML/Schema#Tools)

Concernant la validation, pour un usage occasionnel il existe des services de validation en ligne, dont par exemple :

 <http://www.freeformatter.com/xml-validator-xsd.html>

[\[http://www.freeformatter.com/xml-validator-xsd.html\]](http://www.freeformatter.com/xml-validator-xsd.html)

Pour un usage plus intensif il est possible d'utiliser un plugin pour navigateur ou un **IDE** (*environnement de développement intégré*) dédié à **XML**. Il en existe quelques-uns qui sont gratuits, les plus perfectionnés sont des outils payants.

Pour les développeurs, il existe dans la plupart des langages une **API XML** proposant un parseur et des validateurs pour **DTD** ou **schéma XML**.

### > Exemple

Pour la petite histoire, le mécanisme de validation utilisé dans le cadre de ce cours passe par un programme **PHP** :

```
$xml = new DOMDocument();
$xml->load($full_path);
$schema_path = $xml->documentElement-
>getAttribute("xsi:noNamespaceSchemaLocation");
libxml_use_internal_errors(true);
if ( ! $xml->schemaValidate($schema_path) ) {
    $errors = libxml_get_errors();
    $err_string = "";
    foreach ($errors as $error) {
        $err_string .= get_error($error);
    }
    libxml_clear_errors();
    return $err_string;
}
else return "OK";
```

Ce code est basé sur la classe **DOMDocument** et sa méthode **schemaValidate**, implémentées en **PHP** via la librairie **libxml**.

 La classe **DOMDocument**

[\[http://php.net/manual/fr/class.domdocument.php\]](http://php.net/manual/fr/class.domdocument.php)

 **DOMDocument::schemaValidate**

[\[http://php.net/manual/fr/domdocument.schemavalidate.php\]](http://php.net/manual/fr/domdocument.schemavalidate.php)

 **PHP libxml**

[\[http://php.net/manual/fr/book.libxml.php\]](http://php.net/manual/fr/book.libxml.php)

Ce type de code pourrait très facilement être transposé dans d'autres langages, qui proposent tous le même genre d'**API** pour la validation de documents **XML**.

**N.B.** Dans l'exemple ci-dessus, la mise en forme des messages d'erreurs est obtenue à l'aide de la fonction :

```
function libxml_get_errors($error){
    $return = "";
    switch ($error->level) {
        case LIBXML_ERR_WARNING:
            $return .= "<b>Warning $error->code</b>: ";
            break;
        case LIBXML_ERR_ERROR:
            $return .= "<b>Error $error->code</b>: ";
            break;
        case LIBXML_ERR_FATAL:
            $return .= "<b>Fatal Error $error->code</b>: ";
            break;
    }
    $return .= trim($error->message);
    if ($error->file) {
        $return .= " in <b>".basename($error->file)."</b>";
    }
    $return .= " on line <b>$error->line</b>\n";
    return $return;
}
```