Université Lyon 1

Master CCI

LANGAGES

D'ÉCHANGE

D'INFORMATIONS

L. Médini,

Janvier 2009.

Introduction : objectif général du cours

Comme son titre l'indique, ce cours a pour but de présenter les principaux langages informatiques utilisés pour la description de structures informationnelles plus ou moins complexes. Deux types de structures informationnelles sont abordés dans ce cours : d'une part, les structures documentaires, dont on passera rapidement en revue les différents langages de description, et d'autre part, les données circulant sur Internet, dont la description constituera la majeure partie de ce cours.

Dans les deux cas, l'outil conceptuel utilisé pour la structuration de l'information est la notion de balise. Après avoir donné une définition aussi précise que possible de l'objet manipulé dans ce cours (i.e. l'information), nous définirons ce concept de balise, en termes d'intérêt pour la description des structures de données, de formats génériques dans les différents « langages à balises » et d'utilisation *a posteriori*. Puis nous verrons un historique des différents langages à balises et une description détaillée de certains d'entre eux sera présentée.

1ère PARTIE :

NOTION D'INFORMATION SEMI-STRUCTURÉE ET MÉTHODES DE STRUCTURATION DE L'INFORMATION

1.1 Définition : la notion d'information

La première chose à faire, si l'on veut s'intéresser à la structuration de l'information, est de définir cette notion d'information. De nombreuses définitions existent pour cette notion, chacune étant adaptée à un domaine spécifique d'utilisation (mathématiques, électronique, informatique, mais aussi linguistique, sciences cognitives, journalisme…). Nous en présentons plusieurs définitions, qui abordent cette notion sous des points de vue différents mais néanmoins pertinents pour le domaine des langages à balises informatiques.

1.1.1 L'information en tant que signal

La première définition est fondée sur une description physique de l'information, et constitue la base des différentes disciplines dérivées de la « théorie de l'information », définie par Shannon en 1948. On classe entre autres parmi ces disciplines l'électronique, le traitement du signal, l'ingénierie des réseaux et, accessoirement, l'informatique.

Shannon décrit l'information comme un flux physique circulant entre un émetteur et un récepteur lors d'un processus de communication. D'un point de vue informatique, ces flux peuvent circuler entre :

- Homme – homme : dans ce cas, il s'agit d'un processus de communication non médiatisé par ordinateur (dialogue oral, lettre manuscrite, signaux de fumée…) qui ne nous intéresse pas ici,
- Homme – machine : dans ce cas, le type d'information qui circule est spécifique au périphérique utilisé pour le processus de communication (écran, HP, clavier, souris…) ; définir un tel processus revient à réaliser une IHM[1] ; au niveau le plus simple, les outils utilisés sont les fonctions d'entrées/sorties de base ; il est à noter que dans ce cas, l'information échangée doit être « intelligible » à la fois par l'humain et par le système,
- Machine – machine : ce sont les échanges de données entre programmes, via par exemple une API Windows ou un pipe Unix, ou entre ordinateurs, via les protocoles de communication réseau ; les outils utilisés sont des fonctions spécifiques à ces types de communications.

Dans tous les cas, l'information échangée se conforme à un « protocole de communication », c'est-à-dire à une série de codes qui en définissent la structure [2]. Lorsque l'outil informatique est émetteur ou récepteur de l'information, la mise en forme ou le décodage de l'information sont réalisés par une « couche logicielle ». Nous définissons ici cette couche logicielle comme étant l'« application de production ou d'interprétation » de l'information.

[1] Interface Homme-Machine.

[2] Ces protocoles peuvent être le langage, qui permet de structurer l'information en phrases se référant à une grammaire, ou des codes visuels (comme le soulignement des liens dans les pages web) ou sonores qui permettent à l'utilisateur de distinguer les types d'information qui lui sont présentés, ou encore l'encapsulation des données dans les messages, selon les protocoles IP ou HTTP.

1.1.2 Information et systèmes d'information

Un autre type de classification utilisé pour définir la notion d'information est issu de la théorie des systèmes ou systémique. Cette discipline permet notamment de définir la notion de **Systèmes d'Information** (SI). Dans son acception la plus simple, un système est un objet complexe dont les composants communiquent entre eux et avec l'extérieur.

Même si cette discipline a *a priori* peu de rapport avec l'informatique[3], elle permet d'introduire une dichotomie entre les natures statique et dynamique des composants d'un système. On considère les flux physiques échangés (les données, telles qu'elles sont définies dans le paragraphe précédent) comme statiques et les sous-parties du système qui émettent et reçoivent ces flux (les outils) comme dynamiques. En informatique, cette distinction permet de distinguer les données utilisées par les applications des applications elles-mêmes.

Dans ce cours, la notion de structure informationnelle s'applique par conséquent – presque – uniquement à de l'information statique[4]. Elle concerne les différents langages de balisage comme (X)HTML et XML (voir plus loin). Les outils qui vous sont présentés dans ce cours sont destinés à traiter des structures d'informations fondées sur XML. Elles permettent soit de transformer ces structures, comme XSLT, soit de les parcourir et de les traiter élément par élément, comme le DOM ou SAX. La structure des données et les applications correspondantes ne peuvent pas être décorrélées. Leur interdépendance est à considérer à deux niveaux.

Au niveau production, une application utilisée pour produire différentes structures de données doit être suffisamment générique pour pouvoir supporter la variabilité de ces structures. Si ce n'est pas le cas, toute modification de la structure de données implique le développement d'une nouvelle version de l'application correspondante. Les différentes versions des langages de descriptions de pages Postscript ou PDF, qui ont chacune été accompagnées d'une nouvelle version des applications de production documentaire correspondantes illustrent ce type de problème.

Inversement, une application dédiée à une structure de données trop générique est par nature une « usine à gaz » inutilisable. C'est par exemple la raison pour laquelle, malgré la puissance de ce langage, SGML n'a jamais connu le développement auquel s'attendaient ses concepteurs.

Au niveau interprétation, le format des données reçues par une application – et donc leur structure – doit nécessairement être compréhensible par elle. Cela peut poser des problèmes lorsque l'on ne connaît pas l'application destinataire de l'information. Nous verrons entre autres les problèmes posés par l'interprétation du HTML par les « parseurs »[5] des différents navigateurs.

---

[3] Bien qu'elle soit à la base de certaines méthodes de modélisation en analyse fonctionnelle et en Génie Logiciel.

[4] À l'exception des scripts et applets inclus dans les pages HTML, qui permettent de dynamiser la présentation de ces pages.

[5] Voir plus loin.

1.1.3 Donnée, signal, information et document

Nous avons vu dans les paragraphes précédents que les notions d'information et de donnée se rapportent à des flux statiques échangés entre un émetteur et un récepteur. Cependant, ces termes ne sont pas synonymes. Dans ce paragraphe, nous donnons quelques définitions qui vont nous permettre d'éviter les confusions entre ces termes.

La notion de **donnée** envisage ce flux uniquement sous l'angle de ses caractéristiques physiques. Elle ne peut être décrite qu'à l'aide de descripteurs de forme, comme sa taille, sa nature (flux binaire ou d'octets) ou son type d'encodage (ASCII, Unicode pour du texte, GIF, JPEG pour une image…). La majorité des données que nous traitons dans ce cours étant de type textuel, nous considérons le texte comme type de données par défaut.

La notion de **signal** se distingue de celle de donnée uniquement par un changement de référentiel. Si la donnée se positionne sur le flux lui-même[6], le signal considère ce flux du point de vue du vecteur sur lequel il circule. Parmi les descripteurs d'un signal s'ajoutent donc aux précédents des descripteurs physiques dynamiques (c'est-à-dire temporels), tels que l'intensité pour un signal analogique ou la vitesse de propagation pour un signal numérique.

Comme la donnée, l'**information** se place dans le référentiel du flux, mais s'intéresse à son contenu, et non à ses caractéristiques physiques : l'information véhicule un « sens ». Cette notion de sens est loin d'être évidente, surtout lorsqu'elle est appliquée à des informations circulant entre des machines. Pour simplifier, nous la prendrons ici, comme tous les termes apparentés (sémantique, connaissance…), d'un pont de vue purement pragmatique. L'information véhicule un contenu qui peut être identifié par des descripteurs sémantiques (domaine, mots-clés, « méta-tags »…). Le sens associé à cette information est donc la caractérisation de ce contenu par ces descripteurs.

Un **document** comme ce support de cours est un ensemble cohérent d'informations véhiculant chacune un sens unitaire (on parle d'ailleurs de « grains d'informations »), regroupés autour d'un thème commun. La méthode de regroupement de ces grains d'information en un sens générique est laissée à l'appréciation de l'auteur du document et ne nous intéresse pas pour l'instant.

Physiquement, le document représente une entité informationnelle complète, c'est-à-dire qu'il est l'intégralité du flux échangé. C'est donc sur les documents que nous allons travailler, et non sur les grains d'information isolés. La structure de l'information qui nous intéresse ici est donc l'organisation des différents grains d'information à l'intérieur du document.

1.1.4 Information structurée ou non structurée ?

Dans les domaines documentaires et plus particulièrement en recherche d'information, il est fréquemment fait référence à de l'information structurée et non structurée. Cette distinction oppose les bases de données et les corpus documentaires[7]. Elle concerne non seulement les

[6] D'où son aspect statique : l'ascenseur est immobile dans le référentiel de l'ascenseur.

[7] Pratiquement, un corpus documentaire est un groupe de documents qui sont « indexés » ensemble pour la recherche d'information. Un index est un document sur lequel s'effectue la recherche.

techniques de production et de stockage, mais surtout l'accès au contenu : les techniques mises en œuvre pour retrouver le contenu de chaque champ d'une base de données sont très différentes de celles qui permettent la recherche d'information en texte intégral.

Cependant, si en recherche d'information, il est fréquent d'indexer et de lancer des recherches sur le texte brut des documents, il est erroné d'affirmer que les documents sont dépourvus de structure. Par exemple, tout document textuel qui se respecte possède un titre, une introduction, plusieurs parties (elles-mêmes éventuellement structurées en sous-parties) et une conclusion. Une recherche d'information n'a pas la même signification lorsqu'elle est effectuée sur le titre d'un document que lorsqu'elle a lieu en texte intégral. De la même façon, un document HTML possède une en-tête (« `head` ») et un corps (« `body` »). Les documents XML ont une structure à la fois aussi rigoureuse et plus souple que les bases de données. Le fait de pouvoir considérer les langages à balises à la fois comme des ensembles de grains d'information et comme des structures informationnelles leur confère l'appellation d'**information semi-structurée**. Ils permettent de tirer à la fois avantage des techniques de recherche dans l'information non structurée et d'analyse de l'information structurée. De plus, il existe des mécanismes de transformation de ces structures qui rendent XML utile pour le transfert de données entre les applications.

Il est de plus important de signaler que le sens associé à un document dans un langage à balise est défini à la fois par le contenu des données et par la structure documentaire elle-même. Par exemple, le contenu du titre d'un document peut être plus significatif que celui d'une sous-partie de celui-ci. Mieux, en XML, où l'on est libre de choisir le titre des balises, le nom des balises identifie non plus le type de contenu, mais directement le sens qui y est associé.

Si je veux « ficher » les étudiants d'une promo, je n'ai pas besoin d'employer une structure générique du type :

```
<Titre>Fiche_Promo</Titre>
  <Partie>
    <Sous-titre>Nom</Sous-titre>
    <Texte>Nom étudiant 1</Texte>
    <Sous-titre>Note</Sous-titre>
    <Texte>Note étudiant 1</Texte>
  </Partie>
  <Partie>
    <Sous-titre>Nom</Sous-titre>
    <Texte>Nom étudiant 2</Texte>
    <Sous-titre>Note</Sous-titre>
    <Texte>Note étudiant 2</Texte>
  </Partie>
```

Il me suffit d'écrire :

```
<Fiche_Promo>
  <Etudiant>
    <Nom>Nom étudiant 1</Nom>
    <Note>Note étudiant 1</Note>
  </Etudiant>
  <Etudiant>
    <Nom>Nom étudiant 2</Nom>
    <Note>Note étudiant 2</Note>
  </Etudiant>
</Fiche_Promo>
```

Bien entendu, cette structure n'a de sens que si l'on est capable ensuite d'analyser les éléments `<Etudiant>`, `<Nom>` ou `<Note>`. Nous verrons dans ce cours les différents outils existant autour du langage XML qui permettent l'analyse et la transformation de telles structures de données.

Un autre exemple de l'importance de la structure des données est celui de la balise `<Nom>` dans le descriptif de l'amphi :

```
<Amphi>
  <Prof>
    <Nom>Médini</Nom>
    <Descriptif>Une constante du mouvement</Descriptif>
  </Prof>
  <Etudiants>
    <Nom>…</Nom>
    <Nom>…</Nom>
    <Nom>…</Nom>
    <Descriptif>Varient d'une séance à l'autre</Descriptif>
  </Etudiants>
</Amphi>
```

On voit bien ici que les balises `<Nom>` et `<Descriptif>` sont relatives aux balises qu'elles déterminent.

1.2 Balise, balisage, élément et structure de document

1.2.1 Notion de balise

Une balise est un « signe » particulier, qui permet de repérer une position dans un espace donné. Chaque type de balise est défini en fonction d'un traitement particulier. Par exemple, les balises radio permettent aux avions de s'orienter en leur indiquant la position de certains points au sol, une balise Argos permet de retrouver quelqu'un en signalant sa position, etc.

Dans un document (i.e. un espace linéaire), une balise permet de **marquer** l'emplacement d'une information particulière dans le flux d'information représenté par ce document[8]. Lorsque plusieurs types d'informations différentes doivent être marqués, il est nécessaire d'utiliser plusieurs types de balises. La balise n'est plus alors uniquement un signe dans le document, mais un signe marquant la position recherchée, associé à un élément d'information particulier indiquant le type d'information marqué. **Une balise est donc un élément d'information, au même titre que le contenu du document lui-même.** En cela, on parle donc pour les balises de « méta-information », c'est-à-dire d'information sur le contenu informationnel du document.

En pratique, dans les documents textuels, le signe indiquant une balise est un caractère ou une séquence de caractères particulier(e), comme '/' ou "%%" en langage Postscript, '\' en RTF, '<' en HTML et XML. Le corps de la balise indique ensuite le type d'information repéré.

1.2.2 Balisage de parties de document

Le balisage (littéralement, l'utilisation de balises) permet également de marquer des « zones » (i.e. de parties) d'un document. Cela est par exemple utilisé pour la mise en gras ou en italique d'une partie de texte, ou plus généralement l'application de « styles » dans un texte. En pratique, il s'agit de repérer le début et la fin de la zone spécifique à marquer. On parlera de **balisage explicite** lorsque le début et la fin de cette zone sont identifiés par des balises et de **balisage implicite** sinon. Dans le premier cas, les balises de début et de fin de zones sont appelées **balises ouvrantes** et **balises fermantes**. En règle générale, le balisage implicite conserve les balises ouvrantes et omet les balises fermantes. La fin de la zone balisée est alors repérée par un séparateur simple (comme l'espace ou un signe de ponctuation) ou le début d'une autre balise. L'ensemble de la zone balisée (balise ouvrante, contenu et balise fermante) définit un **élément**.

Un exemple de balisage explicite couramment utilisé est la mise entre parenthèses d'une séquence de texte, pour signifier l'importance moindre d'une zone de texte par rapport au corps de la phrase dont il fait partie. Le découpage d'un texte en phrases est un exemple de balisage implicite encore plus utilisé : des signes de ponctuation comme un point, un point d'exclamation ou d'interrogation sont en fait les balises qui marquent la fin des phrases[9]. En Postscript et en RTF, le balisage explicite utilise des accolades. En HTML, comme dans toute

[8] D'ailleurs, un langage à balises se dit en anglais : « Markup Language », soit langage de marquage.

[9] Le fait que ces phrases ne possèdent pas de balises de début qui pose d'ailleurs des problèmes non triviaux aux outils d'analyse grammaticaux des textes utilisés par exemple dans les moteurs de recherche.

la famille des Markup Languages destinés au web (voir plus loin), les balises fermantes sont identiques aux balises ouvrantes à un caractère près : le slash '/'[10].

En règle générale, le balisage explicite est beaucoup plus facile à « parser » (c'est-à-dire à traiter) que le balisage implicite. C'est pourquoi la plupart des langages de balisage ont choisi un balisage explicite. Le seul cas particulier où le balisage implicite peut être traité simplement est quand un document est constitué d'une suite ininterrompue de zones analogues (comme une suite de phrases dans un texte). Le balisage implicite suppose donc une connaissance *a priori* de la structure du document.

D'un point de vue sémantique, il existe deux grands types de balisages : le **balisage procédural** et le **balisage descriptif**.

Un langage de balisage est dit procédural lorsqu'il permet de définir un **format** de mise en forme (i.e. un « format de sortie ») des documents. Ce type de langage utilise des **conventions sémantiques** qui permettent de spécifier ce format de sortie. Un document décrit dans un langage de balisage procédural est en fait une suite d'instructions de formatage indiquant les opérations à effectuer pour produire le rendu du document souhaité. Pour ces raisons, un langage de balisage procédural est aussi appelé **langage de formatage.**

PS, puis PDF chez Adobe™, RTF et les nombreux formats MS-Word© chez MicroSoft™, HTML et RDF pour les langages issus du W3C, ou encore TeX, sont les langages de balisage procéduraux les plus connus en ce qui concerne les documents textuels.

Un langage de balisage est dit descriptif lorsqu'il se borne à identifier les types d'informations présents dans les documents, sans y associer aucune méta-information de formatage. Dans ce cas, les jeux de balises employés sont spécifiques à ces types d'informations, et varient donc en fonction du type de document décrit. Chaque jeu de balises, ou **Structure Logique Générique (SLG)** correspond à un langage de balisage descriptif, et le sur-ensemble de ces langages est appelé un **méta-langage de balisage**. SGML et XML sont de tels méta-langages.

1.2.3 Différents types de structures documentaires

Lorsqu'un document est fortement structuré (i.e. contient un grand nombre de balises), il est important que le mode de balisage soit cohérent avec sa structure. Il existe plusieurs modèles permettant de représenter la « coexistence » de chacun de ces éléments de structure dans le document. Si le document n'est pas linéaire, mais est par exemple un hyperdocument (i.e. un document hypermédia), cette représentation peut vite devenir très complexe. Nous présentons ici les deux types de décomposition les plus courants.

Les structures arborescentes sont de loin les plus utilisées, car les plus simples à comprendre et à mettre en œuvre d'un point de vue informatique. Dans ce modèle, le document est divisé en parties, elles-mêmes divisées en sous-parties, et ainsi de suite jusqu'à l'ultime élément d'information balisé. C'est sur ce modèle que sont fondés tous les langages de balisage dédiés au web, ainsi que la plupart des langages de description documentaires.

---

[10] Par exemple, en HTML, le corps de la page (i.e. la partie affichée) est délimité par un balisage compris entre les balises `<body>` et `</body>`.

Ce modèle possède cependant un inconvénient de taille : il interdit le chevauchement des divisions d'un document (c'est-à-dire, en pratique, des balises). Dans un pseudo-langage de description arborescent, la mise en forme de la phrase suivante pose un problème.
Voici un exemple en **caractères gras, *gras et italique,*** *puis italique seul,* qui pose problème.

Une représentation simple de cette phrase serait :
```
<phrase>Voici un exemple en <gras>caractères gras, <italique>gras et italique, </gras>puis italique seul, </italique>qui pose problème.</phrase>
```

Malheureusement, le modèle arborescent interdit une telle structure, car toute balise ne peut être enfant que d'une seule balise. Ici, la balise `italique` est enfant tantôt de `gras`, tantôt de `phrase`. Une représentation grammaticalement correcte – mais pas nécessairement satisfaisante – de cette phrase serait :
```
<phrase>Voici un exemple en <gras>caractères gras, <italique>gras et italique, </italique></gras><italique>puis italique seul, </italique>qui pose problème.</phrase>
```

On a donc séparé notre balise `italique` initiale en deux balises, l'une enfant de `gras`, et l'autre de `phrase`. L'inconvénient de ce procédé est qu'il introduit, à l'intérieur d'une zone balisée, une coupure conceptuellement injustifiée. En pratique, si l'on veut modifier le style associé à la mise en italique, on est obligé de le faire pour chacune des deux balises.

D'autre part, j'aurais également pu choisir de représenter cette phrase de la façon suivante :
```
<phrase>Voici un exemple en <gras>caractères gras, </gras><italique>
<gras>gras et italique, </gras>puis italique seul, </italique>qui pose problème.</phrase>
```

voire même :
```
<phrase>Voici un exemple en <gras>caractères gras, </gras><gras_et_italique>gras et italique, </gras_et_italique><italique>puis italique seul, </italique>qui pose problème.</phrase>
```

Ici, on a défini une balise `gras_et_italique` qui « hérite » des caractéristiques des deux autres. Ce choix est conceptuellement plus satisfaisant, mais plus difficile à mettre en œuvre.

Finalement, le choix même d'un mode de représentation d'une telle structure de donnée n'est pas évident. Il importe donc de se préoccuper à l'avance de la façon de les traiter. Heureusement, la grande majorité des documents sont également bâtis sur un modèle arborescent simple, et ne posent pas ce genre de problème.

Les structures relationnelles ou dynamiques permettent de définir des liens entre les différentes parties d'un (hyper)document. L'intérêt de ces structure réside principalement dans le typage des ces relations entre les éléments d'information. Sans rentrer dans les détails, on peut distinguer deux types de liens, comme cela sera mentionné dans la suite du cours : les liens simples et étendus. Les premiers sont équivalents aux liens hypermédias définis en HTML. Les seconds sont traités par des structures de données particulières, comme le langage de description de liens XLink, qui permet de définir des « bases de liens » physiquement dissociées de l'information.

1.3 Historique et positionnement des principaux langages de structuration

Il existe différents types de langages de structuration de l'information. Pour chacune de ces catégories, nous présentons l'historique et l'évolution des standards qui ont permis la mise au point des principaux langages, ainsi que leurs caractéristiques.

1.3.1 Langages de description documentaire (TeX, RTF, PS, PDF),

La fonction documentaire la plus ancienne et la plus répandue est de véhiculer physiquement l'information entre les individus, par exemple sur support papier. Nous présentons ici les principaux langages de description de l'information sous forme de pages, c'est-à-dire permettant de décrire des documents avec leur mise en forme.

Ces langages sont nombreux et ont chacun leurs spécificités. Ils vont du langage utilisé par les logiciels de traitement de texte individuels aux applications professionnelles d'édition et de publication électronique, en passant par les langages de description de calques d'images ou de documents (utilisés par exemple dans Adobe™ Photoshop© ou Acrobat©). Nous nous limitons ici à donner quelques caractéristiques des langages de description de pages « potentiellement textuelles », c'est-à-dire ne se limitant pas à la description d'images.

1.3.1.1 Le langage PostScript

En 1985, la société Adobe Systems Incorporated™ définit le format PS, qui propose une standardisation des formats de sorties vers les imprimantes. L'idée est de s'affranchir de l'installation d'un driver différent pour chaque imprimante, et de définir un langage suffisamment générique pour proposer une description des pages qui incluse toutes les caractéristiques dont ont besoin les différents types d'imprimantes pour « figer » les pages. Avec le développement des serveurs d'impression réseau, l'intérêt est, pour un poste client, de pouvoir gérer la mise en forme des pages à imprimer indépendamment du choix de la ressource où aura physiquement lieu l'impression. Pour cela, le format PS permet la description d'images bitmaps ou vectorielles et du texte formaté, en noir et blanc ou en couleur. Un fichier PS permet de décrire un document composé d'une ou de plusieurs pages formatées.

Malheureusement, du fait des très nombreuses possibilités de ce format, très peu de logiciels sont capables d'interpréter n'importe quel fichier postscript. En particulier, chaque driver d'imprimante implémente une « version » du langage qui lui est propre. Inversement, le rendu d'un fichier peut différer d'une imprimante à l'autre.

Il existe aussi un format EPS (Encapsulated PostScript), qui est paradoxalement un sous-ensemble de PS, puisqu'il ne permet de décrire que des images, et non des pages textuelles. Dans ce format, le document PS d'origine est encapsulé dans une en-tête spécifique à la machine. L'intérêt est également de transmettre des instructions de traitement des pages au périphérique d'impression, en plus de la description de ces pages. Par exemple, il est possible de spécifier dans l'en-tête d'encapsulation que l'on souhaite imprimer un document en recto-verso, ou bien avec deux pages par feuille.

1.3.1.2 Le langage PDF : Portable Document File

Après le succès remporté par le format PostScript, devenu un standard de fait, Adobe™ possède un notoriété suffisante pour se permettre d'« imposer », à partir de 1993, le langage PDF. Pour ne pas se faire de concurrence à elle-même, Adobe positionne le langage PDF sur le marché du document électronique, et continue à promouvoir PS pour celui du document papier. PDF vise donc les échanges de données sur le web ou dans les entreprises, et n'est pas directement interprétable par les imprimantes. Commercialement, Adobe inonde le monde de visualiseurs gratuits (Acrobat Reader®), mais fait payer les outils de génération de fichiers PDF (PDFwriter©, Acrobat Exchange© – jusqu'à la version 3.1 – puis Acrobat Distiller©).

Techniquement, PDF est bâti sur le même principe que PostScript, mais la description des pages est plus stricte que pour son prédécesseur. Celles-ci peuvent alors être imprimées ou visualisées toujours de la même façon, répondant ainsi mieux aux objectifs de portabilité que le langage PS. En particulier, une innovation notable de ce format est de permettre l'inclusion des polices utilisées dans le document. Par ailleurs, pour certains types de documents (i.e. ceux qui ont été scannés puis « océrisés », deux couches sont définies : La première est l'image d'origine et la seconde permet de plaquer des éléments textuels sur cette image, aux endroits où les caractères ont été reconnus.

D'autres caractéristiques du langage PDF sont plus spécifiquement liées au concept de document électronique. Elles visent à faciliter et à sécuriser l'échange de documents sur Internet, en réduisant la taille et en « figeant » ces documents. Pour cela, PDF inclut de la compression de données : un fichier PDF contient une balise de flux (stream) dans laquelle les données de description de la page peuvent être compressées. De plus, la nature de l'offre logicielle fait que même s'il existe des outils permettant la retouche des fichiers PDF, une fois ces fichiers générés, leur modification est fastidieuse.

Plusieurs versions du format PDF ont été proposées, élargissant le spectre des fonctionnalités proposées par ce langage. Il permet maintenant de définir des liens hypermédias dans le document ou vers des ressources distantes, des formulaires dans lesquels l'utilisateur peut entrer des informations ou encore d'indexer et de faire de la recherche en texte intégral sur un corpus de documents PDF.

1.3.1.3 Les langages de traitement de textes

Incontournables, de loin les plus répandus, mais pas nécessairement les meilleurs, les outils de production documentaire (« traitements de textes ») de MicroSoft™ sont tout de même relativement puissants et permettent notamment la rédaction en mode WYSIWYG (what you see is what you get). La preuve en est que j'ai choisi de rédiger ce poly avec Word©, plutôt que d'autres… Les formats associés permettent de stocker à la fois contenu et mise en forme de façon relativement puissante. De plus, l'application de styles ou de modèles de documents permet la définition d'informations de mise en forme génériques et réutilisables. Les langages proposés sont :

- le langage RTF (Rich Text File), qui est le seul standard de fait réellement imposé par MicroSoft (notamment du fait qu'il n'est pas compressé, et qu'un document respectant ce format peut donc « facilement » être généré automatiquement). Le jeu de balises utilise le caractère '/' et les accolades.

- toute la succession de versions du format « .doc », qui s'inspirent de la structure du langage RTF, mais qui l'interprètent chacun différemment, et utilisent des algorithmes de compression.

Sous la pression, la plupart des éditeurs de logiciels proposent des filtres permettant au moins la lecture de fichiers Word, voire la génération de tels fichiers. Les meilleurs de ces filtres ne sont d'ailleurs pas nécessairement ceux de MicroSoft, à en juger par les problèmes de perte de mise en forme constatés à chaque changement de version du format .doc.

À l'opposé de MicroSoft – et par conséquent issu du monde Unix et du domaine des freewares – le langage TeX est l'exemple le plus répandu de langages de production documentaire compilé (exception faite de l'utilisation de XML/XSLT avec un processeur Java, voir plus loin). Créé par Donald E. Knut à la fin des années 1970, ce langage de traitement de texte permet d'écrire du texte « au kilomètre », en incluant les balises de mise en forme directement dans le fichier source, puis de visualiser ultérieurement le résultat des pages obtenues.

1.3.2 Les langages dédiés au web

Avec l'essor du web sont apparus les langages de structuration de l'information spécifiques à ce nouveau média. Leurs caractéristiques diffèrent essentiellement des langages de description de pages du fait que l'information n'est plus destinée à être visualisée sur papier, mais sur écran ou par d'autres canaux, pour lesquels les différents formats papiers ne sont plus des standards. Par ailleurs, l'information n'est plus nécessairement statique, et la notion d'interactivité fait son apparition. Enfin, une prise de conscience des problèmes spécifiques posés par le travail sur écran conduisent à la définition de nouveaux standards et normes (ex : les critères de conception ergonomiques de pages web) prenant en compte ces problèmes.

C'est pour cela qu'a été créé le World Wide Web Consortium, en octobre 1994. Le W3C est l'instance de régulation et de « normalisation » du web. C'est un organisme indépendant qui comprend des chercheurs et des industriels qui travaillent sur les différentes problématiques liées au web dans de nombreux pays[11]. En fait, le W3C ne produit pas de normes, qui restent l'apanage de l'ISO, mais des *recommandations* qui peuvent avoir un caractère normatif. Ces recommandations ainsi que d'autres informations sur les travaux du W3C sont disponibles gratuitement sur le site de cet organisme : http://www.w3.org.

Les principales recommandations du W3C concernent les langages les plus utilisés sur le web, avec en premier lieu HTML, mais aussi XML et toutes ses applications, dont nous aurons l'occasion de reparler. Les travaux du W3C s'articulent autour de sept thématiques ou *activités*, animées par des *groupes de travail*. La figure ci-dessous illustre certains des principaux langages de structuration liés au web et définis par le W3C (à l'exception de SGML, dont la norme a précédé la création du W3C).

[11] Le W3C est principalement basé au Massachusetts Institute of Technology (MIT) aux USA, à l'ERCIM (European Research Consortium for Informatics and Mathematics) sur le site de l'INRIA de Sophia-Antipolis en France et à l'Université de Keio au Japon. Cependant, le W3C est représenté dans une quinzaine d'autres pays dans le monde, et de nombreux partenaires industriels prennent une part active à ses activités. La liste exhaustive des membres du W3C est disponible à http://www.w3.org/Consortium/Member/List.

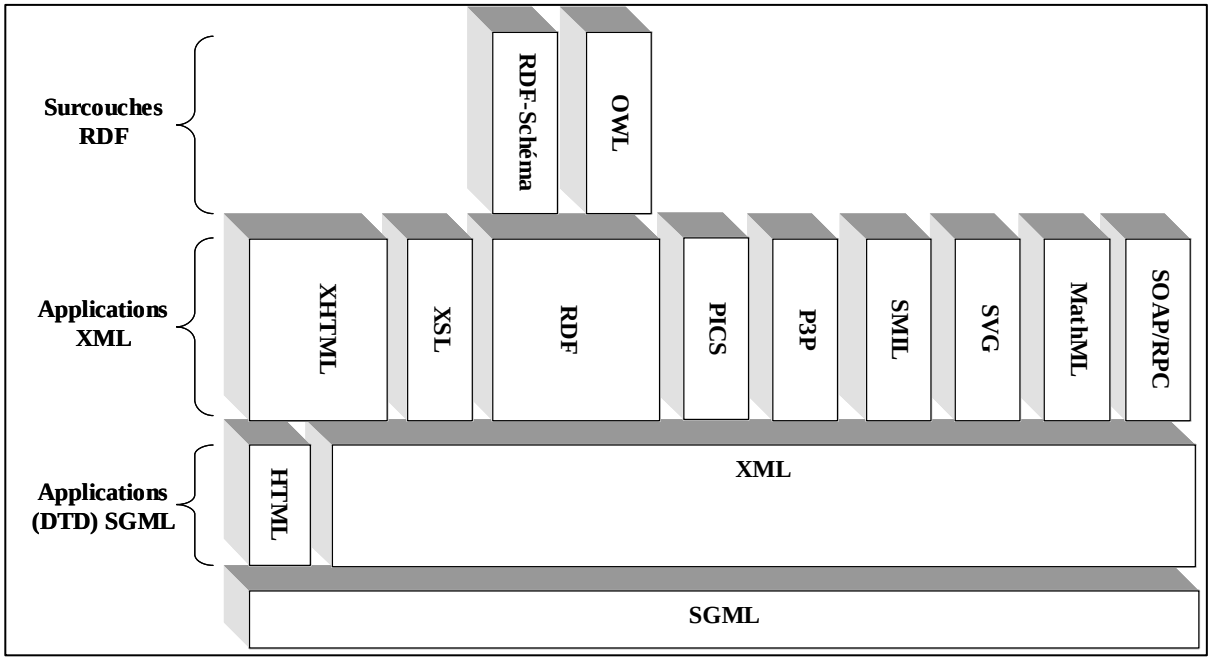


Figure 1. Hiérarchisation des différents langages à balises dédiés au web.

L'énumération ci-dessous présente les principales caractéristiques des langages figurant sur ce diagramme. Certains d'entre eux sont présentés plus en détail dans la partie suivante.

SGML : méta-langage de balisage
début des spécifications à la fin des années 60 ; norme ISO depuis 1986 ;
puissant mais complexe, taille des spécifications imposante (> 500 pages),
- Introduit la notion de DTD (grammaire spécifique d'une application).

HTML : langage de visualisation
- 1ère version de la recommandation du W3C : 1992,
- Version 4.01 : décembre 1997.

XML : méta-langage de structuration
- **Syntaxe et documents XML bien formés** : version 1.0 : février 98 ;
    - **Plusieurs nouvelles éditions** : du 6 octobre 2000 et du 4 février 2004 (mais toujours V1.0),
    - **Version 1.1** : le 4 février 2004.
- **DTD et schémas** : permettent de définir des documents XML valides, c'est-à-dire des syntaxes particulières pour les applications XML.
- **Infosets** : jeux de données abstraits représentant une partie d'un arbre XML. Ces infosets sont constitués d'*information items* représentant des nœuds de l'arbre.
- **Namespaces** : espaces de noms permettant :
    - d'éviter les conflits entre des noms d'éléments similaires,
    - de définir des applications XML,
    - d'identifier des applications existantes.
- **XPOINTER et XLINK** : travaux sur les liens hypermédias
- **Le DOM et SAX** : permettent l'utilisation de XML dans un langage de programmation.

XHTML : version de HTML respectant la syntaxe XML : 1999.

XSL : langage de feuilles de styles spécifique pour XML. Se fonde sur le langage XPATH pour la localisation des nœuds et sur XSLT pour leur transformation.

Ci-dessous, les caractéristiques des autres langages présentés sur la figure précédente, et qui ne seront pas traités dans ce cours :

RDF : Resource Description Framework : langage de description générique des *méta-données* (i.e. des informations sur les contenus des documents), permettant diverses opérations telles que le stockage ou l'échange, indépendamment du type de données. Ce langage est associé à un mécanisme de description de ces méta-données appelé *RDF-Schema*.

OWL : est un langage de description d'ontologies, bâti sur RDF. Une ontologie est une description arborescente permettant de représenter des structures de données complexes, voire les structures de « connaissances », ainsi que les relations ente ces données.

PICS : Platform for Internet Content Selection : permet aux parents et aux enseignants de déterminer les contenus visibles par les enfants sur Internet. Utilisé dans les outils de filtrage.

P3P : Platform for Privacy Preferences Project : projet de d'aide au contrôle par les utilisateurs des informations personnelles qu'ils fournissent sur les sites web qu'ils visitent.

SMIL : Synchronized Multimedia Integration Language : permet d'écrire des présentations multimédias interactives en représentant la temporisation et la synchronisation de plusieurs médias sous forme modulaire dans un langage fondé sur XML. Les composants de SMIL sont utilisés pour intégrer la temporisation dans XHTML et dans SVG.

SOAP : Simple Object Access Protocol : permet de structurer l'échange de données entre applications sur Internet, sous forme de messages simples écrits en XML. Ces messages peuvent en particulier incorporer des appels de procédures distantes (Remote Procedure Call, ou RPC), qui permettent à des applications de contrôler la mise en route et le déroulement d'autres applications à distance.

WSDL : Web Services Description Language : propose un modèle et un format de description des services web fondé sur la syntaxe XML. Ce langage permet de dissocier la description abstraite d'un service de la façon (« où » et « comment ») dont il est implémenté.

Il existe bien entendu d'autres langages à balises, comme CompactHTML, WirelessML (langage de visualisation destiné au WAP), MathML (destiné à la description d'applications mathématiques), Scalable Vector Graphics (qui permet la description d'images vectorielles)… Ces langages ne sont pas non plus traités ici. Cependant, vous pouvez trouver des informations sur l'état actuel d'avancement de chacun d'eux sur le site du W3C.

2ème PARTIE :

APERÇU DE QUELQUES LANGAGES DE STRUCTURATION

2.1 SGML : Standard Generalized Markup Language

À l'origine de tous les langages présentés plus loin se situe le langage SGML. Il succède lui-même à un autre méta-langage, ***GML***, développé chez IBM à partir de 1969. En 1986, SGML devient une norme ISO, qui est amendée en 1988. La norme actuelle est la norme ISO 8879.

SGML est un méta-langage, c'est-à-dire un langage servant à décrire d'autres langages à balises, définis dans un but précis. Ces langages sont appelés des *applications*. Ce sont des langages à balises dont la structure de données est un sous-ensemble de celle autorisée par SGML. Pour définir ces structures de données, SGML s'appuie sur la notion de DTD (Document Type Definition). À chaque application correspond une DTD spécifique, et la DTD est le cœur s'une application SGML. En cela, on assimile souvent un langage à balise et la DTD SGML qui lui correspond. La modification d'une DTD entraîne souvent celle des données qu'elle définit, d'où l'intérêt de porter une attention spéciale à la mise au point de la DTD.

Contrairement aux autres langages présentés plus loin, SGML n'a pas été spécifiquement conçu pour être destiné au web. Au contraire, SGML est « Le » méta-langage de balisage de référence, à partir duquel peuvent être bâties toutes les autres sortes d'applications. Par exemple, d'autres applications de SGML, que nous ne détaillerons pas ici, sont les langages ODA (pour Office Document Architecture), qui est un langage de description documentaire générique, DSSSL et SPDL, qui permettent la description de pages avancée, notamment pour des activités professionnelles comme l'édition, ou HyTime, qui permet de représenter des documents hypertextuels en prenant en compte leurs aspects temporels. Tous ces langages sont des normes ISO.

Si autant d'applications de SGML existent et sont encore utilisées à l'heure actuelle, c'est parce que les concepteurs de ce langage ont volontairement voulu prévoir « toutes » les applications possibles. De plus, SGML a été conçu pour durer. Il affirme prendre en compte les évolutions futures des techniques et des logiciels. Cette puissance et cette évolutivité ont été mises en œuvre au prix d'une grande complexité. La taille des spécifications de ce langage dépasse les 500 pages. De plus, le fait que SGML est une norme ISO rend payant l'accès à ses spécifications. Les prix de vente des outils SGML répercutent bien entendu le coût de l'accès à cette norme.

Finalement, SGML est la solution industrielle : chère et compliquée, mais extrêmement puissante. Par conséquent, même s'il pourrait servir de format d'échange de données pour le web, SGML n'est pas utilisé (à quelques rares exceptions près), car il nécessite un investissement trop lourd en temps, en outils et en formation.

2.2 HTML et XHTML

HTML (HyperText Markup Language) est une application (i.e. une DTD) de SGML, destinée à visualiser sur écran des hyperdocuments constitués de « pages web ». Chacune de ces pages est un *document HTML*, dont la syntaxe respecte les prescriptions de la DTD HTML. La syntaxe de ce langage, ainsi que la façon dont les documents HTML doivent être visualisés sont définies dans les recommandations du W3C : http://www.w3.org/TR/HTML. La version courante du langage HTML est la version 4.01. Actuellement, le W3C semble considérer cette version comme figée, puisque cette version n'a pas été actualisée depuis 1997.

Le successeur de HTML est le langage XHTML, qui utilise les mêmes balises que HTML, mais avec une syntaxe XML et des contraintes de mise en forme plus strictes. La recommandation XHTML date de 1999 et est disponible à l'adresse http://www.w3.org/TR/XHTML.

Dans ce paragraphe, nous décrivons la structure d'un document HTML, les différences fondamentales entre ce langage et XHTML, et les principales balises HTML 4.01 / XHTML 1.0 et leurs caractéristiques de visualisation avec le navigateur Internet Explorer de MicroSoft™. Les langages de feuilles de style CSS, associés à HTML et XHTML sont ensuite présentés.

Bien entendu, cette présentation n'est pas exhaustive et ne garantit pas la portabilité des documents générés. L'emploi de HTML s'est généralisé avec l'essor du web, et les documents HTML peuvent être visualisés par de nombreux navigateurs, qui donnent parfois des interprétations différentes des balises HTML. Néanmoins, elle suffit pour commencer à générer des pages web avec une mise en forme acceptable, et peut servir de base pour la génération dynamique de pages web virtuelles, via les langages XML et XSL présentés plus loin.

2.2.1 La syntaxe HTML et XHTML

Dans ce paragraphe, nous décrivons de façon synthétique les règles de syntaxe de ces deux langages, leurs ressemblances et leurs différences, de façon à pouvoir en aborder le contenu dans le paragraphe suivant.

2.2.1.1 Balisage HTML

La DTD HTML est un peu lâche sur la définition des balises. La seule règle à appliquer est d'entourer chaque balise d'un chevron ouvrant '<' et d'un chevron fermant '>'. Deux chevrons ouvrants ne peuvent pas se succéder dans un document HTML. HTML accepte à la fois des balises encadrant un contenu, ou balises *conteneur*, et les balises vides, ou balises *autonomes*. Dans le premier cas, le contenu est entouré d'une balise ouvrante, contenant un **nom de balise** éventuellement suivi d'**attributs**, et d'une balise fermante, contenant un slash '/' et le nom de la balise à refermer. Dans le second cas, seule une balise d'ouverture est requise.

À chaque nom de balise est associé l'un de ces deux types de balises par la DTD. Le navigateur « sait » alors, en fonction de ce nom, s'il doit chercher une balise de fin ou non.

Les noms de balises sont des mots pouvant inclure des lettres et des chiffres, prédéfinis par la DTD HTML. Ils peuvent être écrits en majuscules ou minuscules.

Les attributs sont des zones de texte particulières incluse dans le corps des balises ouvrantes. Le mode de traitement par le navigateur de chaque attribut par rapport à une balise donnée est défini dans la recommandation du W3C. En HTML, un attribut a obligatoirement un nom et éventuellement une valeur. Cette valeur doit être du texte simple ; si elle contient des espaces, elle doit être écrite entre guillemets (simples ou doubles[12]). Ils peuvent être écrits en majuscules ou minuscules.

Les commentaires sont encadrés par la séquence de caractères `<!--` et `-->`.

Voici des exemples de balises HTML valides, avec ou sans contenus, avec ou sans attributs :

```
<P>Exemple de balise conteneur de texte simple (paragraphe)</P>

<p>Exemple de balise conteneur de texte simple (paragraphe) <b>et
d'une balise de mise en gras</b></P>

<!-- Exemple de balise vide : --> <HR>

<P ALIGN=center>Exemple de balise avec attribut valué</p>

<OPTION SELECTED>Exemple de balise avec attribut non valué</Option>

<Body onload="javascript :alert('chargement de la page') ;">
 <H1>Exemple d'imbrication de guillemets</h1>
</body>
```

2.2.1.2 Balisage XHTML

Bien qu'il ait été créé exactement dans le même but que HTML – à savoir : visualiser des pages web – le langage XHTML n'est pas une DTD de SGML, mais une application XML (voir §2.4.3). Par conséquent, XHTML respecte la syntaxe XML, qui est présentée en détail au §2.4.4. C'est la raison pour laquelle la syntaxe XHTML est plus stricte que celle de HTML.

Au niveau du contenu, XHTML reprend la quasi-totalité des balises HTML 4.01. Les quelques règles suivantes devraient vous permettre de passer sans effort vos documents HTML en XHTML.

- Tous les noms de balises et d'attributs, ainsi que les valeurs prédéfinies des attributs doivent être écrits en minuscules.
- Toutes les balises conteneurs doivent être correctement fermées, sans chevauchement d'autres balises.
- Toutes les balises vides doivent posséder le caractère slash '/' avant le chevron de fermeture.

---

[12] L'utilisation de deux types de guillemets différents permet l'imbrication de chaînes de caractères dans la valeur d'un atttribut, en imbriquant l'un des types dans la valeur d'attribut délimitée par l'autre.

- Chaque attribut doit avoir une valeur (et une seule).
- Toutes les valeurs d'attributs doivent être placées entre guillemets.
- L'identification d'un élément se fait à l'aide de l'attribut `id`, alors que c'était l'attribut `name` qui était utilisé en HTML. Pour des raisons de compatibilité, il est conseillé d'affubler les éléments litigieux des deux attributs, avec la même valeur.
- Les caractères de contrôle (voir §2.2.1.3) ne doivent jamais être utilisés tels quels, même dans les scripts ou les feuilles de style internes aux documents XHTML. Ils doivent être remplacés par des références aux entités qui les désignent. La solution la plus propre reste cependant de placer les scripts et feuilles de style dans des fichiers séparés, où la syntaxe XHTML n'a pas cours.

La recommandation XHTML 1.0 définit trois « dialectes » : **frameset, transitional**, et **strict**.

- Comme son nom l'indique, le premier se limite à la définition de pages contenant des cadres, et n'est pas très intéressant. Les deux autres sont utilisés pour définir les contenus visualisables des pages.
- La différence entre le XHTML strict et transitionnel se situe au niveau de la façon dont les informations de mise en forme sont indiquées dans le document. Le XHTML transitionnel est le plus souple : il accepte les pages contenant des balises ou des attributs de formatage, tels que `<b>` ou `align="center"`, pourvu qu'elles respectent la syntaxe du langage XHTML telle qu'elle est décrite ci-dessus.
- Le XHTML strict reprend l'esprit XML, qui est fondé sur le principe d'une séparation du contenu informationnel et de la mise en forme. C'est pourquoi une page conforme à la recommandation XHTML-Strict ne doit présenter que la structure du document, sans aucun formatage. La mise en forme doit en être séparée, et placée dans une feuille de style, interne ou externe. Les deux éléments de code précédents sont interdits en XHTML, et doivent être remplacés par l'application de styles CSS. Une balise `<b>` HTML devient par exemple : `<span class="gras">`.

Autant que faire se peut, on tentera de respecter la syntaxe XHTML stricte. Elle est également valide en HTML, nettement plus lisible et beaucoup plus pertinente en termes de cohérence et de maintenance des styles. Par exemple, l'utilisation de feuilles de styles CSS externes permet d'harmoniser sans effort les différents cadres d'une page, et de répercuter immédiatement toute modification d'un style sur l'ensemble d'un site.

2.2.1.3 Références aux entités caractères

Le jeu de caractères pris en charge par défaut dans la spécification HTML est le jeu ANSI (UTF-8 ou UTF-16) américain. Il n'intègre pas de nombreux caractères utilisés dans les langues non anglophones, comme les caractères accentués ou les cédilles. D'autre part, certains caractères ne peuvent pas apparaître dans le texte d'une page HTML, comme le caractère inférieur '<', qui symbolise l'ouverture d'une balise.

Pour afficher tous ces caractères, HTML fait appel à la notion d'**entité**. Une entité est l'équivalent d'une variable de type char en C, qui permet de donner un nom aux caractères que l'on veut afficher. Pour cet affichage, on remplace dans le texte de la page HTML, le caractère par une **référence** à cette entité. Une telle référence est de la forme `&Nom_Entité;`. Il existe deux types de noms d'entités : les noms de référence des normes ISO spécifiques aux jeux de caractères correspondants et les références via les numéros de caractères Unicode. Le

tableau suivant donne la liste de quelques entités du jeu de caractères ISO Latin1 et des caractères de contrôle, avec leurs noms ISO et Unicode :

Caractères ISO Latin1			Noms ISO	Noms Unicode	Caractères
Noms ISO	**Noms Unicode**	**Caractères**	agrave	#224	à
Agrave	#192	À	…	…	…
Aacute	#193	Á	egrave	#232	è
Acirc	#194	Â	…	…	…
Atilde	#195	Ã	igrave	#236	ì
Auml	#196	Ä	…	…	…
Aring	#197	Å	ograve	#242	ò
AElig	#198	Æ	…	…	…
Ccedil	#199	Ç	ugrave	#249	ù
Egrave	#200	È	…	…	…
Eacute	#201	É	yuml	#255	ÿ
…	…	…	**2.2.1.3.1.1 Caractères de contrôle**		
Igrave	#204	Ì	amp	#38	&
…	…	…	apos	#39	'
Ograve	#210	Ò	gt	#62	>
…	…	…	lt	#60	<
Ugrave	#217	Ù	quot	#34	"
…	…	…		#91	[
Yacute	#221	Ý		#93	]

2.2.1.4 Chevauchement des éléments

La DTD HTML interdit à des balises conteneur de se chevaucher (voir §**Erreur ! Source du renvoi introuvable.**). Les réactions des navigateurs aux pages enfreignant cette règle varient d'ailleurs en fonction des balises, des navigateurs et même de leurs versions. Certains ont tendance à ignorer la deuxième balise de fermeture (c'est le cas des anciennes versions de Netscape Navigator®[13]), d'autres à refermer implicitement la balise la plus interne et à ignorer la balise de fermeture mal placée, et d'autres encore à supporter le chevauchement, en fermant la balise interne, et en la rouvrant juste après la balise de fermeture externe. Si nous prenons l'exemple des balises de mise en forme `<b>` (bold) et `<i>` (italic), le code suivant :

```
Texte sans mise en forme, <b>Texte en gras, <i>Texte en gras et italique, </b>Texte en italique, </i>Texte sans mise en forme.
```

apparaîtra, en fonction des cas, de l'une des trois façons suivantes :

Texte sans mise en forme, **Texte en gras, *Texte en gras et italique,*** *Texte en italique, Texte sans mise en forme.*

[13] Ce défaut a d'ailleurs été corrigé dans les versions récentes, pour pouvoir visualiser correctement le HTML généré automatiquement par MS Word©. En effet, s'étant aperçu de ce « problème », les concepteurs de certaines versions de Word ont fait en sorte d'inverser systématiquement les balises fermantes des styles employés, de façon à ce qu'Internet Explorer® soit le seul outil capable de visualiser les pages générées correctement.

Texte sans mise en forme, **Texte en gras,** ***Texte en gras et italique,*** Texte en italique, Texte sans mise en forme.

Texte sans mise en forme, **Texte en gras,** ***Texte en gras et italique,*** *Texte en italique,* Texte sans mise en forme.

2.2.2 Structure d'un document (X)HTML

Un document HTML ou XHTML est constitué de deux balises.

- Une balise de description de type de document. Une telle balise a la forme suivante : `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">`. Nous reviendrons ultérieurement sur la forme de la déclaration de type de document. Techniquement, une telle balise sert à identifier la DTD SGML qui permet de valider la structure du document. C'est donc une balise SGML et non HTML. C'est pourquoi elle peut être placée à l'extérieur de la balise suivante. Dans le cas d'un document XHTML, la déclaration de type de document est une déclaration XML. Elle diffère selon la version de XHTML choisie.
    - Pour un document XHTML-Strict :

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

    - Pour un document XHTML-Transitional :

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

    - Pour un document XHTML-Frameset :

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

- Une balise de conteneur HTML délimitée par une balise d'ouverture : `<html>` pour un document HTML (ou `<html xmlns="http://www.w3.org/1999/xhtml">` pour un document XHTML) et une balise de fermeture `</html>`. Toutes les autres balises HTML doivent être comprises à l'intérieur de celles-ci. L'ensemble de ces éléments (X)HTML est contenu dans deux ou trois balises de structures :
    - `<head>` définit l'en-tête d'un document HTML dans lequel seront stockées toutes les informations et méta-informations non affichées à l'écran, telles que le titre du document, son auteur, les éventuelles propriétés de style, les scripts ou des champs d'en-tête HTTP particuliers.
    - `<frameset>` définit une séparation de la page en cadres. Dans le cas où une balise `<frameset>` est présente, la balise `<body>` est optionnelle, et ne sert qu'à définir des contenus destinés aux navigateurs n'acceptant pas les cadres.
    - `<body>` définit le corps du document, où sont situées les informations directement visualisables, avec les balises de mise en forme.

Le tableau suivant décrit les contenus de ces éléments.

BALISES FONDAMENTALES	
`<html>…</html>`	Délimite le document HTML.
`<head>…</head>`	Délimite l'en-tête du document HTML.

<frameset>…</frameset>	Définit le jeu de cadres d'une fenêtre ou d'un cadre.
<body>…</body>	Délimite le corps du document HTML.
< !--…-->	Commentaire.
BALISES D'EN-TÊTE	
<base />	Définit une valeur de référence globale pour le document. Utilisé avec un et un seul des attributs ci-dessous.
href="…"	URL de référence, qui servira de base au traitement des URL relatives..
target="…"	Cadre cible des liens par défaut.
<meta>…</meta>	Permet d'insérer de l'information non affichée par le navigateur (méta-information).
name="…"	Type de méta-information. Exemple de valeur : keywords.
content="…"	Contenu de la méta-information. Exemple : liste de mots-clés.
<meta>…</meta>	Permet de spécifier la valeur d'un champ d'en-tête HTTP.
http-equiv="…"	Nom du champ d'en-tête. Exemple de valeur : Refresh (réactualise la page après un délai défini).
content="…"	Contenu du champ d'en-tête. Exemple de valeur : 5 ; URL=http://www.monsite.com (5 est le temps entre les rafraîchissements, suivi de l'URL où aller chercher la page).
<link />	Lie le document à une ressource externe (par exemple une feuille de style). Utilisé dans <head>.
href="…"	Adresse de la ressource.
rel="…"	Type lien (stylesheet pour les feuilles de style).
type="…"	Type de contenu internet (text/css pour les feuilles de style CSS).
<style>…</style>	Permet d'inclure une feuille de style CSS interne dans un document.
<script>…</script>	Permet de définir un script pour rendre une page HTML dynamique.
<title>…</title>	Indique le titre du document.
BALISES DE DÉFINITION DE CADRES	
<frameset>…</frameset>	Définit le jeu de cadres d'une fenêtre ou d'un cadre.
rows="…"	Séparation des cadres horizontale. Utilisé dans <frameset>. Valeur : les largeurs des cadres séparées par des virgules. Unité : points (px) ou pourcents (%).
cols="…"	Séparation des cadres verticale. Utilisé dans <frameset>. Valeur : les largeurs des cadres séparées par des virgules. Unité : points (px) ou pourcents (%).
<frame />	Identifie chaque cadre de <frameset> dans l'ordre où ils sont définis.
src="…"	Nom du fichier HTML source du cadre.
id="…"	Identificateur du cadre. Permet d'y faire référence depuis un autre cadre.
BALISES DE CORPS DE DOCUMENT : BALISES DE STRUCTURATION DE TEXTE	
…	Titre de 1er niveau.
…	Titre de 2ème niveau.
…	Titre de 3ème niveau.
<h4>, <h5>, <h6>	Titre de nième niveau. Pas de mise en forme prédéfinie.
…	Paragraphe.
 	Saut de ligne.
<hr />	Ligne horizontale.
id="…"	Identifiant d'un élément quelconque du document XHTML. Doit être unique.
name="… "	Nom d'un élément du document HTML. Utilisé pour la compatibilité ascendante avec les navigateurs non compatibles XHTML. Il est conseillé d'utiliser les deux attributs avec la même valeur.
…	Ne fait rien, définit l'emplacement d'attributs style ou autres.
…	Identique à , mais insère un saut de ligne en fin de balise.

class="…"	Permet d'appliquer un style défini dans une feuille de style CSS (interne ou externe). Utilisable dans , ,<body> et dans la plupart des autres balises HTML. **Fortement recommandé en XHTML.**
style="…"	Contient les définitions de style intégrés avec la syntaxe CSS. Utilisable dans , ,<body> et dans la plupart des autres balises HTML. **Toléré par la recommandation XHTML-Strict**.
BALISES DE CORPS DE DOCUMENT : TABLES	
<table>…</table>	Définit une table.
<th>…</th>	En-tête de table. Mise en forme particulière du contenu. Utilisé dans <table>.
<tr>…</tr>	Ligne de table. Utilisé dans <table>.
<td>…</td>	Colonne de table. Utilisé dans <tr>.
border="…."	Épaisseur de la bordure de la table. Valeurs : 0 (défaut)= invisible ; 1, 2 etc.
BALISES DE CORPS DE DOCUMENT : FORMULAIRES	
<form>…</form>	Délimite l'emplacement d'un formulaire.
method="…"	Type de méthode utilisée pour l'envoi du formulaire. Valeurs autorisées : get, post. Valeur par défaut : get.
action="…"	Action à réaliser à la soumission du formulaire.
<button>…</button>	Introduit un bouton standard dans la page.
<input>…</input>	Balise générique d'entrée dans un formulaire.
type="…"	Type de l'entrée standard. Valeurs autorisées : text, hidden, checkbox, radio, submit, reset.
checked="checked"	Indique qu'un élément de type radio (bouton radio) est coché par défaut.
<select>…</select>	Liste déroulante.
<option>…</option>	Élément de la liste déroulante. Utilisé dans <select>.
selected="selected"	Indique qu'une option de la liste est sélectionnée par défaut.
<textarea>…</textarea>	Zone de texte.
BALISES DE CORPS DE DOCUMENT : IMAGES	
<img… />	Identifie le point d'insertion d'une image.
src="…"	URL permettant d'accéder au fichier contenant l'image.
alt="…"	Info-bulle (pop-up) où on peut faire apparaître des informations sur l'image.
BALISES DE CORPS DE DOCUMENT : ANCRES DE LIENS	
<a>…	Avec l'attribut href, définit le point d'ancrage d'un lien vers un autre document ; avec l'attribut name, définit un point d'ancrage vers lequel un lien peut pointer.
href="…"	URL du document ou du point d'ancrage vers lequel pointe le lien.
title="…"	Info-bulle (pop-up) où on peut faire apparaître des informations sur le lien.
BALISES DE CORPS DE DOCUMENT : AUTRES OBJETS	
<applet>…</applet>	Applet Java.
code="…"	Fichier java compilé de l'applet (.class).
<object>…</object>	Objet de type quelconque.
param="…"	Définition d'un paramètre de l'objet.
BALISES DE CORPS DE DOCUMENT : LISTES ET CARACTERES SPÉCIAUX	
…	Liste à puces classée(numérotée).
…	Liste à puces non classée.
start="…"	Valeur initiale de numérotation de cette liste (balise).
type="…"	- Type de numérotation ; valeurs possibles A, a, I, i et 1 (défaut). - Type de puce utilisé pour marquer les éléments de la liste ; valeurs possibles : disc, circle et square.
…	Élément de liste utilisé avec et .
value="…"	Valeur numérique de cet élément de liste (affecte cet élément et tous les éléments suivants dans les listes).
<dl>…</dl>	Liste de définitions
<dt>…</dt>	Terme de définition appartenant à une liste de définitions.
<dd>…</dd>	Définition correspondant à un terme de définition, appartenant à une liste de définition. Contenu décalé vers la droite.

BALISES DE CORPS DE DOCUMENT : MISE EN FORME DU TEXTE (HTML et XHTML transitionnel seulement)	
…	Gras.
…	Italique.
<u>…</u>	Souligné.
<strike>…</strike>	Barré petit (trace une ligne sur le texte).
…	Petit texte.
<big>…</big>	Gros texte.
[…]	Exposant.
_…	Indice.
<tt>…</tt>	Petite police télétype à chasse fixe.
<pre>…<pre>	Petite police télétype à chasse fixe. Conserve les espaces et les sauts de ligne.
BALISES DE CORPS DE DOCUMENT : POLICES (HTML et XHTML transitionnel seulement)	
…	Définition d'une police.
size="…"	Taille de la police, de 1 à 7. La taille par défaut est 3. Peut également être spécifiée comme valeur relative à la taille actuelle, par exemple,+2 ou-1.
color="…"	Change la couleur du texte.
face="…"	Nom de la police utilisée si elle se trouve dans le système de l'utilisateur. Des virgules peuvent séparer plusieurs noms de polices et la première police utilisée dans la liste sera utilisée.
BALISES DE CORPS DE DOCUMENT : ATTRIBUTS DE STYLE (HTML et XHTML transitionnel seulement)	
align="…"	Alignement horizontal du texte. Utilisable avec <p>,<h1>,<h2> etc. Valeurs autorisées : left, center, right.
valign="…"	Alignement vertical du texte d'un élément de table. Utilisable avec <tr>,<td>. Valeurs autorisées : top, middle, bottom.
height="…"	Hauteur d'une zone d'écran. Utilisable avec , <table> ou <applet>.
width="…"	Largeur d'une zone d'écran. Utilisable avec , <table> ou <applet>.

2.3 Les feuilles de style CSS

Pour la composition de documents au format W3 (et *a fortiori* de documents XHTML), le standard recommandé par le W3C est le langage CSS (Cascading Style Sheets). Il existe plusieurs versions de ce langage. CSS1, dont la recommandation du 17 décembre 1996, révisée le 11 janvier 1999, est disponible à l'adresse http://www.w3.org/TR/REC-CSS1 spécifie la syntaxe de base de la sélection et de la déclaration de règles de style et des mécanismes d'héritage entre ces règles (« mise en cascade ») utilisé depuis HTML.

CSS2 (recommandation du 12 mai 1998, http://www.w3.org/TR/REC-CSS2) ajoute à CSS1 la spécification de feuilles de style dépendantes du support (par exemple, les feuilles de styles d'impression ou aurales) et la prise en charge des polices de caractères téléchargeables, du positionnement d'éléments, de tables…

CSS3 est en cours de spécifications. Un aperçu de l'évolution de ses spécifications est disponible à l'URL : http://www.w3.org/Style/CSS/current-work. Ce cours ne traite que les versions 1 et 2 du langage CSS. Il est librement inspiré du cours de CSS de N. Chu, disponible sur le web : http://www.nc-technologies.com/fr/css-intro.asp.

2.3.1 Principes fondamentaux

Une feuille de style CSS est un élément informationnel destiné à spécifier les caractéristiques de mise en forme du document (X)HTML, à partir des balises contenues dans ce document. L'intérêt est d'utiliser les balises HTML uniquement pour hiérarchiser le contenu du document et les feuilles de style CSS pour spécifier leur mise en forme. Par exemple, à une balise `<h1>`, est attaché un niveau de titre, pour lequel on peut choisir, pour tout le document, voire pour un site entier, la police, la taille, la couleur sans faire appel aux attributs de mise en forme HTML.

Remarques :

- Théoriquement, on devrait parler de feuille de styles au pluriel, puisque l'on définit un style pour chaque type de balise. Toutefois, l'orthographe au singulier étant la plus fréquemment rencontrée, nous nous conformons à cette pratique.
- On parle également de feuilles de style en cascade, car dans le cas où plusieurs feuilles de style peuvent s'appliquer à la même balise, un ordre de priorité est déterminé par le navigateur pour leur application. Cette notion est à distinguer de celle d'héritage des propriétés de style. Nous reviendrons plus loin sur ces deux notions, qu'intègre le langage CSS.

Précisons enfin que les feuilles de style ne sont pas une composante du langage HTML mais un langage à part dans la famille des Markup Languages du W3C.

Les avantages de l'utilisation de feuilles de style CSS par rapport aux attributs de style HTML sont les suivants :

- Séparation du contenu et de la mise en forme.
- Cohésion de la présentation de sites grâce aux feuilles de style externes.
- Facilité de la maintenance de la mise en forme d'une page ou d'un site sans en modifier le contenu.
- Syntaxe différente de celle de HTML :

    - plus logique, car créée spécifiquement pour la définition d'attributs de style[14].
    - peut être incluse sans ambiguïté dans un document (X)HTML.
- Amélioration de la prise en charge des styles par rapport à HTML (surtout depuis CSS2) : contrôle des polices, de la distance entre les lignes, des marges, des indentations, positionnement au pixel près du texte et/ou des images, feuilles de style auditives…
- Harmonisation de la syntaxe des propriétés de style et de leur prise en charge par les différents navigateurs. Cette harmonisation est cependant encore loin d'être effective ; vous en aurez quelques exemples dans ce chapitre.

2.3.2 Modes de définition des styles CSS

Il existe 3 manières d'utiliser les feuilles de style dans un document HTML. Ces feuilles de style peuvent être **intégrées**, **incorporées**, ou **liées**. Ces méthodes ont chacune leurs avantages et leurs particularités.

2.3.2.1 Styles intégrés

Un style intégré permet de fixer localement des attributs de style à une partie de document. Il s'insère directement dans la balise dont il définit le contenu. On ne peut donc pas parler dans ce cas de feuille de style, sauf à considérer l'ensemble des définitions de styles éparpillées dans le document. La déclaration d'un style intégré se fait dans la valeur d'un attribut `style`. La syntaxe à utiliser est décrite au paragraphe 2.3.3 :

```
<html>
 <body>
  <h1 style="font-family: Arial; font-style: italic">
   Titre en Arial italique
  </h1>
 </body>
</html>
```

2.3.2.2 Feuilles de style incorporées

Une feuille incorporée est déclarée dans l'en-tête du document (X)HTML. Elle s'applique globalement à toutes les balises du document pour lesquelles elle définit des styles.

```
<html>
<head>
<style type="text/css">
<!--
Définition des styles
-->
</style>
</head>
<body>
Balises auxquelles sont appliqués les styles
</body>
</html>
```

[14] En HTML, on a souvent recours à des « artifices de style » (tables imbriquées, layers) pour obtenir la présentation souhaitée, sans garantie qu'elle soit identique pour tous les navigateurs. Le code devient alors vite assez fouillis, et inconsistant d'une page à l'autre.

Remarques :
- La balise `<style>` introduit la définition d'une feuille de style.
- L'attribut `type="text/css"` indique (sous forme de type MIME) que ce qui suit est du texte et qu'il s'agit de cascading style sheets (CSS). Pour l'instant, il s'agit de la seule possibilité mais on peut prévoir à l'avenir d'autres versions de ce langage.
- Les balises de commentaires `<!-- ... -->` empêchent les navigateurs qui ne savent pas traiter les feuilles de style d'essayer d'interpréter ces instructions comme du code HTML. En pratique, la quasi-totalité des navigateurs est capable d'interpréter ces instructions, ce qui rend les commentaires inutiles, voire piégeux : certaines versions récentes des navigateurs (d'Internet Explorer par exemple) ignorent la définition d'une feuille de style si celle-ci est placée dans un commentaire HTML. Il n'est donc pas nécessairement recommandé de placer la définition d'une feuille de style entre commentaires, quoi qu'en dise la recommandation du W3C.

2.3.2.3 Feuilles de style liées

Une feuille liée est contenue dans un fichier séparé, possédant une extension `.css`, et sera traitée par le navigateur comme une ressource distante autonome. Outre une dissociation physique du fond et de la forme, le principal intérêt de l'utilisation de feuilles de style liées est qu'une même feuille de style CSS peut être liée à plusieurs pages web, et ainsi assurer la cohérence graphique d'un site. De plus, ne faisant pas partie de la page (X)HTML, elle n'en subit pas les restrictions, notamment au niveau de l'emploi des caractères de contrôle (qui, pour mémoire, doivent impérativement être traités par référence en XHTML Strict).

Pour réaliser une feuille liée, on place toutes les feuilles de style dans un fichier CSS, par exemple `styles.css`. Aucune en-tête particulière n'est nécessaire pour ce type de fichiers. Le contenu de ce fichier pourrait par exemple être :
```
td {font-family : Arial ; font-size : 10pt ;}
```

On insère ensuite une référence au fichier CSS dans la balise `<head>` de la page web concernée, en utilisant la balise `<link>`, décrite au chapitre précédent :
```
<link rel="stylesheet" type="text/css" href="styles.css" />
```

Rappels :
- La balise `<link>` indique au navigateur que la page est liée à une autre ressource.
- L'attribut `rel="stylesheet"` précise qu'il y trouvera une feuille de style externe.
- L'attribut `type="text/css"` précise le type MIME du fichier.
- L'attribut classique de lien `href` donne le chemin d'accès et le nom du fichier à lier.

2.3.3 Syntaxe du langage

Dans les exemples présentés plus haut, nous avons vu qu'il existe deux syntaxes distinctes. Pour les styles intégrés, la définition du style est directement indiquée dans la valeur de l'attribut (X)HTML `style`. Il n'y a donc pas besoin d'indiquer à quel(s) élément(s) du document le style doit s'appliquer. Dans les deux autres cas, la feuille de style précise le **sélecteur**, c'est-à-dire la balise ou la classe (voir plus loin) à laquelle le style va s'appliquer, et la définition du style entre accolades.

La définition d'un style est une suite d'**instructions CSS**, séparées par des points-virgule. Chaque instruction permet de définir une **propriété de style** par la valuation d'un **attribut de style**. Une propriété de style est l'entité conceptuelle que représente en CSS un attribut de style. Comme il n'est question ici que du langage CSS, et comme toutes les propriétés sont reliées à un et un seul attribut CSS, on utilisera indifféremment ces deux termes pour définir la syntaxe du langage. Une instruction CSS a la syntaxe suivante :

```
nom de l'attribut de style : valeur de l'attribut de style.
```

Au final, on obtient les syntaxes suivantes :

- pour les styles intégrés s'appliquant à des balises non vides :

```
<balise style="attribut : valeur; attribut : valeur ; ...">
 contenu de la balise
</balise>
```

- pour les styles intégrés s'appliquant à des balises vides :

```
<balise style="attribut : valeur; attribut : valeur ; ..." />
```

- pour les feuilles liées ou incorporées :

```
Sélecteur(s) {attribut : valeur; attribut : valeur ; ...}
Sélecteur(s) {attribut : valeur; attribut : valeur ; ...}
...
```

Remarques :

- Les attributs de style pour lesquels la définition d'une valeur est autorisée varient en fonction des types de balises. Par exemple, il est inutile de définir une police pour une balise image.
- Les instructions CSS incorrectes ou inadéquates par rapport au type de balise sont ignorées par le navigateur.
- Le nombre des instructions contenues dans la définition d'un style est illimité.
- Les espaces entre attributs de style et valeurs ne sont pas obligatoires mais aident fortement à la lisibilité du code source. Il en va de même pour celles situées entre deux instructions.
- Pour la lisibilité toujours, vous pouvez écrire vos styles sur plusieurs lignes.
- Pour les commentaires à l'intérieur des feuilles de style, on utilisera la notation du C :
  `/* commentaires */.`
- Dans certains cas, on peut attribuer plusieurs valeurs à une même propriété de style. On séparera alors les différentes valeurs par des virgules :
  `H3 {font-family: Arial, Helvetica, Times New Roman}`
  Dans cet exemple, on indique au navigateur que l'on souhaite utiliser une police Arial. Si cette police n'est pas connue, il doit alors se rabattre sur une police Helvetica, ou à défaut utiliser du Times New Roman, et ainsi de suite.
- On peut attribuer un même style à plusieurs balises (séparées par des virgules) :
  `h1, h2, h3 {font-family: Arial; font-style: italic}`

2.3.4 Les différents types de sélecteurs

On appelle **sélecteurs simples** les balises HTML classiques, auxquelles on attribue des caractéristiques de style dans les feuilles de style CSS. Les styles définis avec ces sélecteurs simples sont communs à toutes les balises du même type. Il est alors difficile de modifier spécifiquement le style de certaines de ces balises.

2.3.4.1 Notion de classes

Pour résoudre ce problème, CSS définit la notion de **classe de style**. Une classe est un **sélecteur contextuel** qui associe un style particulier à un élément en fonction de sa situation (i.e. de son type de balise).

Une définition de style utilisant un sélecteur contextuel a la forme :
```
balise.nom_de_classe { propriété de style: valeur}
```

La mention de la balise est facultative. On peut aussi définir une classe indifféremment d'un type de balise :
```
.nom_de_classe { propriété de style: valeur }
```

Dans ce cas, l'emploi du point (`.`) devant le nom de classe est indispensable.

Pour appliquer le style à une balise, on utilise un attribut `class` :
```
<balise class="nom_de-classe"> .... </balise>
```

Par exemple, si l'on souhaite mettre ce qui est important dans le texte en gras et en bleu, on crée une classe `Important` :
```
.Important { font-weight: bold; font-color: #000080 }
```

Dans le document (X)HTML, on peut appeler la classe depuis n'importe quelle balise :
```
<h1 class="Important">Document contenant du texte important </h1>
<p>Texte normal</p>
<p class="Important">Texte important</p>
<p>Texte normal</p>
<table>
 <tr>
  <td>Cellule normale</td>
  <td class="Important">Cellule importante</td>
 </tr>
</table>
```

2.3.4.2 Notion de pseudo-classe

Une pseudo classe est un ensemble d'éléments (X)HTML qui répondent à un même critère de **contexte de forme**. Un contexte de forme est un état particulier dans lequel se trouve un élément du document à un moment donné. On peut alors définir pour chacun des états de la balise une mise en forme particulière, en utilisant la syntaxe suivante :
```
balise:contexte_de_forme {instructions CSS}
```

Concrètement, les pseudo-classes ne s'appliquent qu'à la balise d'ancre de lien `<a>`. Cette balise peut en effet connaître plusieurs contextes de formes, qui sont énumérés ci-dessous :

`a:link`	Le lien n'a pas encore été visité.
`a:visited`	Le lien a déjà été visité.
`a:active`	Le lien est en train d'être visité (l'utilisateur a cliqué sur le lien et le navigateur attend une réponse du serveur).
`a:hover`	L'utilisateur pointe sur le lien, mais n'a pas cliqué dessus.

Exemple:

```
a:link {color:blue;}
a:visited {color:magenta;}
a:active {color:red;}
a:hover {color:red; text-decoration:none; font-weight:bold;}
```

2.3.4.3 Sélecteurs ID

On peut aussi utiliser les identifiants (attributs `id` des balises, possédant tous des valeurs différentes) pour leur appliquer des styles. Les sélecteurs ID fonctionnent exactement comme les classes, mais sont précédés du caractère dièse (#) :

```
#nom_d'ID {Instructions CSS }
```

Il n'y a alors pas besoin de spécifier explicitement que l'on souhaite appliquer le style. Cela est fait automatiquement lorsque le navigateur rencontre une balise de la forme :

```
<balise id="nom_d'ID"> .... </balise>
```

L'intérêt de ce sélecteur par rapport à une classe qui ne serait appelée qu'une fois dans le document n'est pas évidente à première vue. En fait, il ne permet que de gagner un attribut dans la balise d'ouverture d'un élément (X)HTML. On tâchera donc de ne pas l'utiliser.

2.3.5 Héritage et cascade

2.3.5.1 L'héritage en CSS

CSS intègre la notion d'héritage des propriétés de style : tout élément (X)HTML enfant d'un autre élément hérite des propriétés de style définies pour cet élément parent. Cette règle subit cependant deux limitations :

- Il faut que ces caractéristiques soient applicables à l'élément enfant : nous verrons plus loin qu'il existe deux grands types d'éléments (X)HTML (bloc et en-ligne), qui ne peuvent pas recevoir les mêmes caractéristiques de style.
- Il faut que l'héritage de la propriété ait un « sens » : par exemple, certaines propriétés graphiques, comme le positionnement d'un élément de bloc ne sont pas héritées par défaut, car cette propriété n'a d'intérêt que lorsqu'elle est définie spécifiquement pour chaque élément.

Dans l'exemple ci-dessous, nous faisons hériter le style défini pour la balise `<H1>`, à une balise enfant `<I>` :

```
<H1 style="font-weight:bold">
 Le principe de <I> l'héritage </I>
</H1>
```

Le résultat affiché sera :

Le principe de *l'héritage*

Remarque : dans cet exemple, nous nous sommes permis de mettre les balises en majuscules, puisque le code est nécessairement en HTML et en non XHTML. En effet, la balise `<i>` n'existe pas en XHTML.

Remarques :

- La définition d'une propriété de style pour une balise, dans un style intégré ou dans une feuille de style, annule tout mécanisme d'héritage de cette propriété.

- Nous avons vu plus haut que certaines propriétés de style, comme les caractéristiques de positionnement graphique, ne sont pas héritées. Si l'on veut forcer cet héritage, CSS permet d'utiliser la valeur d'attribut : `inherit`. Cette valeur n'est cependant pas indiquée dans la liste des propriétés du §40, car elle ne fait pas partie des valeurs les plus communément utilisées du langage.

2.3.5.2 Les feuilles de style en cascade

En cas de concurrence entre plusieurs éléments de style, intervient la notion de **cascade** ou d'ordre de priorité. La concurrence entre plusieurs éléments de style provient des différentes possibilités de localisation de feuilles de style :

- dans un fichier CSS externe.
- dans la balise `<head>` du document.
- dans les attributs `style` des balises concernées.

La règle de priorité définie par la spécification CSS est d'appliquer le style défini de la manière la plus spécifique à chaque élément. Ainsi, un style intégré sera retenu par rapport à un style déclaré dans une feuille incorporée, lui-même prioritaire par rapport à celui d'une feuille liée.

Il y a cependant moyen de contourner ces règles de priorité par la déclaration `! important;`

Par exemple, avec le code :

```
<head>
<style type="text/css">
body { background-color : #0000FF }
</style>
</head>
<body style="background-color:#FF0000; ">
...
```

Le fond d'écran sera rouge (#FF0000). Par contre, si l'on change la définition du style appliqué à l'élément `<body>` avec le code :

```
body { background-color : #0000FF ! important;}
```

Le fond d'écran sera bleu (#0000FF).

Remarque : cet exemple ne fonctionne qu'avec Internet Explorer.

2.3.6 Les éléments et

Ces balises sont utilisées pour délimiter les parties du document auxquelles on veut appliquer les styles. Elles permettent de regrouper des éléments ou au contraire d'isoler des parties d'éléments non repérés par des balises (X)HTML structurelles. Elles permettent d'appliquer les caractéristiques des deux grands groupes d'éléments (X)HTML : les éléments de **bloc** et les éléments **en-ligne**.

Les premiers permettent de définir des « zones d'écran » qui peuvent recevoir des caractéristiques de positionnement, de forme et de mise en forme du contenu. Les éléments `<table>`, `<h1>`, `<h2>`, etc. ou `<p>` sont des éléments de blocs.

Les seconds délimitent des portions du « flux de document », pour lesquels on ne peut définir que la mise en forme du contenu. Les balises de mise en forme HTML, telles que <I>, <B> ou <STRONG> sont des exemples d'éléments en-ligne.

2.3.6.1 La balise

 est une balise de bloc qui permet de définir une division de la page. Ce bloc est de forme rectangulaire, et est précédé et suivi d'un saut de ligne. Il peut recevoir des caractéristiques de **mise en forme bloc**, liées à sa forme, comme la hauteur, la largeur ou l'alignement du texte à l'intérieur du bloc, ou des caractéristiques liées à sa position ou à la mise en forme de son contenu. L'exemple suivant présente l'utilisation de l'élément pour la définition d'un style :

```
<html>
 <head>
  <style type="text/css">
   p{ text-align:left}
   .zone{color:red; width:100%; text-align:center}
  </style>
 </head>
 <body>
  <p>ici, le texte est aligné à gauche</p>
  <div class="zone">
   Ici, le texte est centré et affiché en rouge
  </div>
 </body>
</html>
```

Remarque : Netscape ne supporte pas l'imbrication de (et plus généralement des balises de bloc). Les en cascades sont donc à éviter si vous ne savez pas à l'avance avec quel type de navigateur vos pages seront visualisées.

2.3.6.2 La balise

 est une balise de marquage en-ligne qui permet de délimiter une portion de texte. À l'inverse de , elle n'insère pas de saut de ligne et n'a pas de forme particulière. Elle ne peut donc recevoir que des caractéristiques de style dites de **mise en forme en ligne**. Elle sert le plus souvent à définir les caractéristiques du texte qu'elle contient.

Par exemple, le code suivant :

```
<html>
 <head>
  <style type="text/css">
   .element {font-weight:bold}
  </style>
 </head>
 <body>
  <p>Voici du texte en <span class="element">gras</span>.</p>
</body>
</html>
```

Affiche la mise en forme suivante :

Voici du texte en **gras**.

Remarque : en fait, `<span>` ne doit pas impérativement contenir que du texte. Elle peut contenir toutes sortes d'autres éléments (X)HTML. Toutefois, l'intérêt de la balise `<span>` n'est pas flagrant dans ce cas, car cette balise ne sert en général qu'à recevoir un attribut `class` ou `style`, qui peut alors très bien être inséré directement dans ces éléments enfants.

2.3.7 Propriétés de style graphiques

Grâce aux feuilles de styles CSS, il est possible de positionner du texte ou une image au pixel près. Le positionnement des éléments par les feuilles de style est repris sous la spécification CSS-P représentant le positionnement dynamique, inclus dans la recommandation CSS2. Le positionnement dynamique est un ensemble d'attributs de style, destinés à la gestion de la position, des paramètres de visualisation et de superposition des balises de blocs (X)HTML.

Ce positionnement n'est pris en charge qu'à partir des versions 4 de Netscape et d'Internet Explorer. Cette technique pose d'ailleurs des problèmes de compatibilité entre les navigateurs. De plus, si Internet Explorer accepte des attributs CSS-P sur à peu près toutes les balises, Netscape est plus capricieux et a souvent tendance à ignorer ces attributs. De manière générale, il est conseillé de ne les employer que pour des balises de blocs `<div>`.

2.3.7.1 L'attribut `position`

`position` spécifie le type de positionnement du bloc dans le document. Elle peut – entre autres[15] – prendre les valeurs suivantes : `static` (valeur par défaut), `absolute`, ou `relative`.

Une position en flux normal `{position: static}` signifie que le bloc est placé en fonction de sa position dans l'ordre des balises du document. Sa position ne peut pas être modifiée dynamiquement.

Une position absolue `{position: absolute}` est déterminée par rapport aux bords de la fenêtre du navigateur. Pour cela, cette propriété est accompagnée de la spécification des coordonnées du bloc, à l'aide des attributs de localisation `top`, `right`, `bottom` et `left`. Il y a 3 types de valeurs pour ces attributs :

- pas de valeur particulière (`auto`) : valeur de la propriété par défaut,
- valeur en pixels (suivie de `px`),
- valeur en pourcentage (suivie de `%`).

Par exemple, la balise :
```
<div style="position: absolute; top:0px; right:0px}>
```
sera affichée dans le coin supérieur droit de la fenêtre du navigateur.

Remarques :

[15] Se référer à la spécification CSS2 pour toutes les valeurs possibles.

- En positionnement absolu, il est possible de placer un élément en dehors de la fenêtre, soit en donnant des valeurs négatives aux attributs de localisation, soit en leur donnant des valeurs plus grandes que la taille de l'écran.
- En modifiant dynamiquement les valeurs des attributs de localisation (à l'aide de fonctions JavaScript par exemple), on peut déplacer des éléments sur la page.

Une position relative `{position: relative}` est traitée dans le flux du document (c'est-à-dire de bas en haut), et fait référence pour son positionnement à l'élément (X)HTML qui lui est immédiatement supérieur. Les attributs de localisation sont les mêmes et s'utilisent de la même façon que pour la position absolue.

2.3.7.2 L'attribut `clip`

Cet attribut définit une **zone de rognage**, c'est-à-dire la partie du contenu rendu d'un élément qui est visible. Par défaut, la zone de rognage a la même taille et forme que la boîte, ou les boîtes, de l'élément. Cependant, celle-ci peut être modifiée à l'aide de l'attribut `clip`. Cet attribut s'applique uniquement aux blocs en position absolue. Les valeurs les plus utiles pour cet attribut sont `auto` (valeur de la propriété par défaut) et `rect(haut,droite,bas,gauche)`. Dans le deuxième cas, il s'agit de spécifier les distances en pixels par rapport aux bords du bloc. Les valeurs négatives sont admises. L'exemple suivant permet de définir une bordure autour d'une image.

```
<html>
 <head>
  <style type="text/css">
   .toto { position:absolute; top:10px; left:10px;
           width:100px; height:70px;
           clip:rect(10, 25, 10, 25);
           background-color: blue;}
   img { width:50px; height:50px;}
  </style>
 </head>
 <body>
  <div class="toto"><img src="1.jpg" /></div>
 </body>
</html>
```

Remarque : cette propriété a l'avantage d'être traitée de la même façon par Internet explorer et par Netscape : elle n'est actuellement prise en charge par aucun de ces deux navigateurs. Je la laisse dans ce cours, au cas où…

2.3.7.3 L'attribut `visibility`

Cet attribut précise si un élément est affiché à l'écran ou non. Il peut prendre deux valeurs, qui sont différentes sous Internet Explorer et sous Netscape :

- `visible` pour Explorer, `show` pour Netscape,
- `hidden` pour Explorer, `hide` pour Netscape.

La valeur par défaut est `visible`/`show`.

2.3.7.4 L'attribut `display`

Cet attribut est semblable à l'attribut `visibility`. Il s'applique à tous les types d'éléments (X)HTML et est traité – correctement – par Internet Explorer et Netscape. Il peut prendre les valeurs suivantes :

- `block` : l'élément est affiché comme un élément de type bloc, c'est-à-dire séparé des éléments précédent et suivant par des sauts de ligne,
- `inline` : l'élément est affiché en fonction de son type (bloc si c'est un élément de type bloc, en-ligne sinon),
- `none` : l'élément ne fait pas partie du flux d'affichage du document. Les éléments précédent et suivant sont affichés l'un à la suite de l'autre[16].

La valeur par défaut est `inline`.

2.3.7.5 L'attribut `z-index`

Z-index permet d'indiquer une coordonnée imaginaire de « profondeur d'empilement », qui définit l'ordre de superposition des blocs. Il s'applique à tous les éléments de type bloc. Plus l'index est grand, plus l'élément est situé devant les autres. Deux éléments de même index se superposent dans leur ordre d'arrivée dans le flux de document. La valeur par défaut de z-index est 1.

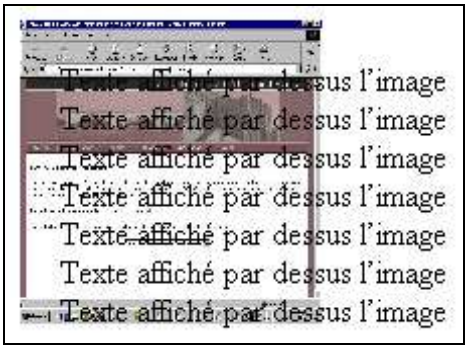
Remarque : cette propriété définit uniquement un ordre de superposition des blocs, et non une quelconque idée de rendu graphique de distance entre ces blocs ou de perspective.

Par exemple, le code suivant :

```
<html>
 <head>
  <style type="text/css">
  .dessous { position:absolute; top:10px; left:10px; }
  .dessus { position:absolute; top:30px; left:30px; z-index:2; }
  </style>
 </head>
 <body>
  <div class="dessous"><img src="1.jpg"></div>
  <div class="dessus">
   Texte affiché par dessus l'image<br />
   Texte affiché par dessus l'image<br />
   ...
  </div>
 </body>
</html>
```

donne :

[16] À la différence de `visibility: hidden`, qui conserve un espace réservé pour le bloc, mais n'affiche rien à l'intérieur.



2.3.8 Couleurs

En CSS comme en HTML, il existe deux façons de spécifier une couleur : en utilisant l'une des 16 couleurs standard définies dans la recommandation HTML 4.0, identifiée par son nom, ou en la composant vous-même.

2.3.8.1 Les couleurs HTML standard

Valeur	Couleur	Valeur	Couleur
`aqua`	Bleu vert	`navy`	Bleu marine
`black`	Noir	`olive`	Vert olive
`blue`	Bleu	`purple`	Pourpre
`fuschia`	Rose fuschia	`red`	Rouge
`gray`	Gris	`silver`	Gris argent
`green`	Vert	`teal`	Bleu gris
`lime`	Vert acidulé	`white`	Blanc
`maroon`	Maron	`yellow`	Jaune

2.3.8.2 Les couleurs RGB

Il est possible de composer soi-même une couleur en utilisant la méthode additive et en spécifiant une intensité pour chacune des trois couleurs fondamentales (rouge, vert et bleu). Ces intensités sont données en hexadécimal et sont codées sur un octet. L'ensemble de la couleur composée est introduit par le caractère dièse (#) et s'appelle le codage RGB de la couleur. Exemples :

- `#000000` : noir
- `#FFFFFF` : blanc
- `#FF0000` : rouge
- `#00FF00` : vert
- `#0000FF` : bleu
- `#00FFFF` : cyan
- `#FF00FF` : magenta
- `#FFFF00` : jaune

2.3.9 Liste des propriétés CSS

La liste suivante présente brièvement les principaux attributs de style CSS, l'effet des propriétés qu'ils implémentent, les valeurs les plus utilisées de ces attributs et des exemples

d’utilisation. Ces spécifications sont tirées de la recommandation CSS2 du W3C, et sont données sans garantie de leur implémentation dans les navigateurs.

2.3.9.1 Styles de police

`font-family`

Nom de police (une police précise : Arial, Times, Helvetica...) ou une famille de police (serif, sans-serif, cursive, fantasy, monospace)

```
h3 {font-family: Arial}
```

`font-style`

Style du texte : `normal`, `italic` ou `oblique`

```
h3 {font-style: italic}
```

`font-weight`

Épaisseur de la police : `normal`, `bold`, `bolder` ou `lighter` ou valeur numérique (`100` | `200` | `300` | `400` | `500` | `600` | `700` | `800` | `900`)

```
p {font-weight: bold}
```

`font-size`

Taille de la police : `xx-small`, `x-small`, `small`, `médium`, `large`, `x-large`, `xx-large`, `larger` ou `smaller` ou taille précise en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage de la police parente(`%`).

```
p {font-size: 12pt}
```

`font-variant`

Modification de la police normale : `normal` ou `small-caps`

```
p {font-variant: small-caps}
```

`font`

Raccourci pour les différentes propriétés de police.

```
p {font: bold italic}
```

2.3.9.2 Styles de texte

`text-align`

Alignement du texte : `left`, `center` ou `right`.

```
h1 {text-align: center}
```

`text-indent`

Retrait dans la première ligne d'un bloc de texte. Spécifié en inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
p {text-indent: 1cm}
```

`text-decoration`

Mise en forme particulière du texte (barré, clignotant, etc.) : `blink`, `underline`, `line-through`, `overline` ou `none`.

```
a:visited {text-decoration: blink}
```

`text-transform`

Modification de la casse du texte : `uppercase` (met les caractères en majuscules), `lowercase` (met les caractères en minuscules) ou `capitalize` (met le premier caractère en majuscule).

```
p {text-transform: uppercase}
```

`color`

Couleur du texte : nom de couleur HTML 4.0 ou code RGB hexadécimal.

```
h3 {color: #000080}
```

`word-spacing`

Espacement des mots : valeur en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
p {word-spacing: 5pt}
```

`letter-spacing`

Espacement des lettres : valeur en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`), ou pourcentage (`%`).

```
p {letter-spacing: 2pt}
```

`line-height`

Interligne (*i.e.* espace entre les lignes du texte) : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
p {line-height: 10pt}
```

`width`

Largeur d'un élément de bloc : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`), ou pourcentage (`%`).

```
h1 {width: 200px}
```

`height`

Hauteur d'un élément de bloc : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`), ou pourcentage (`%`).

```
h1 {height: 100px}
```

`white-space`

Espace ou blanc : `normal`, `pre` ou `nowrap`.

```
pre {white-space: pre}
```

2.3.9.3 Arrière-plans

`background-color`

Couleur de l'arrière-plan : nom de couleur HTML 4.0, code RGB hexadécimal ou `transparent`.
```
body {background-color: #000000}
```

`background-image`

Image d'arrière-plan : URL de l'image.
```
body {background-image: Images/image1.gif}
```

`background-repeat`

Nombre de répétitions de l'image d'arrière-plan : `repeat` (par défaut), `no-repeat`, `repeat-x` (l'image ne se répète qu'horizontalement) ou `repeat-y` (L'image ne se répète que verticalement).
```
p {background-image: image.gif; background-repeat: repeat-x}
```

`background-attachment`

Spécifie si l'image d'arrière-plan reste fixe avec les déplacements de l'écran : `scroll` ou `fixed`.
```
body {background-image: image.gif; background-attachement: fixed}
```

`background-position`

Position de l'image d'arrière-plan par rapport au coin supérieur gauche de la fenêtre ; spécifiée par un couple de valeurs séparées par une espace : {top ou center ou bottom ET left ou center ou right}, ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).
```
body {background-image: img.gif; background-position: right top}
```

`background`

Raccourci pour les différentes propriétés d'arrière-plan.
```
p {background: image.gif fixed repeat-y}
```

2.3.9.4 Marges

`margin-top`

Taille de la marge supérieure : `auto` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).
```
td {margin-top: 5px }
```

`margin-right`

Taille de la marge droite : `auto` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`p {margin-right: 5px }`

`margin-bottom`

Taille de la marge inférieure : `auto` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h1 {margin-bottom: 5px }`

`margin-left`

Taille de la marge gauche : `auto` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h3 {margin-left: 5px }`

`margin-left`

Taille de la marge pour les quatre côtés à la fois : `auto` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h5 {margin: 5px }`

2.3.9.5 Bordures et "enrobages"

`border-top-width`

Épaisseur de la bordure supérieure : `thin` ou `medium` ou `thick` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h3 {border-top-width: thin}`

`border-right-width`

Épaisseur de la bordure droite : `thin` ou `medium` ou `thick` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h3 {border-right-width: medium}`

`border-bottom-width`

Épaisseur de la bordure inférieure : `thin` ou `medium` ou `thick` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h3 {border-bottom-width: thick}`

`border-left-width`

Épaisseur de la bordure gauche : `thin` ou `medium` ou `thick` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (%).
`h3 {border-left-width: 0.5cm}`

`border-width`

Épaisseur de la bordure des quatre côtés à la fois : `thin` ou `medium` ou `thick` ou en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h4 {border- width: 5px}
```

`border-color`

Couleur de la bordure : nom de couleur HTML 4.0 ou code RGB hexadécimal.

```
h3 {border-color: yellow}
```

`border-style`

Style du trait de la bordure : `none`, `solid`, `dotted`, `dashed`, `double`, `groove`, `ridge`, `inset` ou `outset`.

```
table {border-style: solid dashed}
```

`border`

Regroupe toutes les propriétés de bordure.

`padding-top`

Remplissage entre le haut du contenu de l'élément et le bord : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h3 {padding-top: 3px}
```

`padding-right`

Remplissage entre la droite du contenu de l'élément et le bord : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h3 {padding-right: 3px}
```

`padding-bottom`

Remplissage entre le bas du contenu de l'élément et le bord : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h3 {padding-bottom: 3px}
```

`padding-left`

Remplissage entre la gauche du contenu de l'élément et le bord : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h3 {padding-left: 3px}
```

`padding`

Remplissage entre le contenu de l'élément et le bord des quatre côtés à la fois : en points (`pt`), inches (`in`), centimètres (`cm`), pixels (`px`) ou pourcentage (`%`).

```
h5 {padding: 3px}
```

2.3.9.6 Listes

`list-style-type`

Type de puces ou de numérotation : `disc`, `circle`, `square` `decimal`, `lower-roman`, `upper-roman`, `lower-alpha` ou `upper-alpha`.
`ol {list-style-type: square}`

`list-style-image`

Remplacement des puces par une image : URL ou `none`.
`ol {list-style-image: image.gif}`

`list-style-position`

Spécifie si les puces sont à l'intérieur ou à l'extérieur du texte : `inside` ou `outside`.
`ul {list-style-position: inside}`

`list-style`

Regroupe toutes les propriétés de liste.

2.4 XML : Extensible Markup Language

2.4.1 Définition

XML est un méta-langage de description des données, c'est-à-dire qu'il permet de définir des langages de description d'informations structurées, encore appelés langages de structuration, par opposition aux langages de transformation (i.e. de programmation). Par abus de langage, on dira souvent que XML est lui-même un langage de structuration, car les descriptions des données engendrées utilisent la syntaxe XML.

En soi, XML ne sert à rien (au sens applicatif du terme). En revanche, il est possible, à partir d'une de ces descriptions des données, d'utiliser les nombreuses applications de XML pour « faire quelque chose » avec ces données. Par exemple : affichage en HTML, traitements automatiques des données (extraction de données, tri , transformation…)

XML permet de faire apparaître clairement les données (l'information) dans une structure de données (méta-information).

En cela, XML est également indiqué pour l'échange des données entre les applications (format pivot). Il peut aussi être utilisé pour l'échange entre les individus, du fait qu'il est au format texte et que sa structure est « lisible ». Mais dans le cas d'informations textuelles, nous considérons que les individus échangent des documents mis en forme (données + méta-données + informations de formatage), plutôt que XML (données + méta-données uniquement).

2.4.2 Positionnement de XML par rapport à SGML

XML est fondé sur SGML. Il découle d'une volonté de simplification des spécifications de ce langage. XML est un sous-ensemble de SGML, possédant les mêmes objectifs que SGML : la spécification d'applications de structuration des données, mais de manière plus simple et avec des spécifications plus compactes (44 pages pour la version française de la recommandation XML contre environ 500 pour SGML)[17].

En SGML, ces applications ne sont pas clairement identifiées. On s'y réfère uniquement par le nom de leur DTD. En XML, la notion d'espaces de noms permet d'identifier les applications. Ces applications sont appelées *langages de balisage* depuis XML.

Le premier document de travail du W3C date de novembre 1996, et la première édition de la version 1.0 a été publiée le 10 février 1998. Plusieurs éditions ont été publiées depuis, corrigeant des erreurs typographiques mineures. La dernière date de janvier 2004. Jusqu'à cette date, la recommandation XML 1.0 n'a pas évolué, et cette stabilité est due à la simplicité de sa syntaxe.

---

[17] Par conséquent tout ce qui respecte les spécifications d'XML respecte celles de SGML, mais la réciproque n'est pas vraie. Par exemple, SGML définit la notion de balise de façon beaucoup plus large que XML : elles sont en général encadrées par des chevrons ("<" et ">"), mais cela n'est pas imposé dans la spécification du langage. Il pourrait s'agir d'accolades ("{" et "}") comme de toute autre paire de caractères. XML n'accepte pas autre chose que les chevrons pour encadrer les balises.

Cependant, une version 1.1 de la recommandation XML a été publiée le 4 février 2004, introduisant pour la première fois des changements dans son contenu. Le principal de ces changements réside dans le jeu de caractères autorisés dans les noms de balises : en XML 1.0, tous les caractères qui ne sont pas explicitement autorisés sont interdits. En XML 1 .1, c'est le contraire : la recommandation présente une liste exhaustive des caractères interdits (pour des raisons techniques) et autorise tous les autres. Cette liste utilise le jeu de caractères Unicode qui est une extension internationale du code ASCII.

On peut donc en théorie écrire des noms de balises XML dans sa langue nationale. Cependant, du fait de la « jeunesse » de cette recommandation, très peu d'outils prennent actuellement en charge les jeux de caractères étendus dans les noms de balises. C'est pourquoi ce cours se limite à XML 1.0 et à ses applications, dont les autres caractéristiques sont identiques, et restent valables en XML 1.1.

En tout état de cause, la dernière édition de la recommandation en vigueur est disponible à l'adresse : http://www.w3.org/TR/REC-xml.

2.4.3 Les différents composants de XML

Nous avons vu que XML est un méta-langage de balisage. Plusieurs « entités » lui sont associées pour permettre la mise au point, la lecture ou la visualisation de l'information structurée.

XML est la spécification de la syntaxe du langage de structuration utilisé pour la description des données. Cette spécification permet d'écrire des *documents XML bien formés*. Ces documents XML comprennent l'information et la méta-information à structurer.

Les *déclarations de type de document* (DTD) ou les *schémas XML* permettent de définir la structure de l'information décrite de façon plus ou moins approfondie. Lorsqu'un document XML est associé à une DTD ou à un schéma XML et qu'il est conforme à la description de l'information indiquée, on dit qu'il est *valide*.

Un processeur (ou *parser*) XML est une application qui permet d'analyser et de traiter l'information contenue dans les fichiers XML. Il existe des parsers validants et non validants (on dit aussi validateurs et non validateurs), selon qu'ils analysent ou non la conformité du document avec une DTD ou un schéma XML éventuellement associés. La simplicité de la spécification d'XML par rapport à SGML rend également l'écriture d'un parser plus simple pour XML que pour son prédécesseur. La plupart des parsers sont écrits en C/ C++ ou en Java. Pour des raisons pratiques, nous utiliserons dans les TD le parser MSXML de MicroSoft™ qui est inclus dans Internet Explorer depuis la version 5.00. Ce parser permet d'analyser et d'afficher directement les documents XML dans le navigateur, même sans qu'aucune feuille de style ne leur soit associée.

Le modèle objet de document (*DOM*) permet à l'application d'accéder aux données d'un document XML. Il s'interface entre le parser et l'application. Les spécifications du DOM sont définies par niveaux. Celles des niveaux 1 et 2 sont définies, et le W3C travaille actuellement sur DOM niveau 3. Le DOM définit des *interfaces* qui permettent d'accéder aux objets (éléments) d'un document XML. Une interface propose des propriétés et des méthodes pour chaque type d'élément. Il existe un DOM commun pour toutes les applications, le *DOM*

Core) et des extensions : DOM XML, DOM HTML[18], DOM CSS… La combinaison langages de scripts / DOM / HTML constitue le Dynamic HTML (*DHTML*).

L'API simplifiée pour XML (Simple API for XML), ou *SAX* est une alternative au DOM. Elle est adaptée à l'analyse de documents XML volumineux et est plus complexe à mettre en place, car elle nécessite la connaissance de la programmation en Java et l'installation d'un parser (xp) et d'un environnement de développement (jdk) spécifiques. SAX ne sera pas utilisée dans ce cours.

Les espaces de noms (*namespaces*) permettent de définir des « catégories » de vocabulaires XML. Ils sont notamment utilisés pour éviter les conflits entre des termes identiques ayant des significations différentes dans des domaines différents. L'utilisation d'espaces de noms permet de traiter ces termes sans connaître la structuration ni les méta-données du document.

Par exemple, sur une carte de visite, le titre d'une personne correspond à une fonction professionnelle (docteur, ingénieur…) ; pour un document, le titre est la désignation générique du document. Un document de type carte de visite peut comporter ces deux sortes de titres, qu'il est important d'arriver à différencier.

Il existe des « espaces de noms qualifiés » déterminés par la recommandation *Qname* du W3C.

***XLink* (XML Linking Language), XML Base et *XPointer* (XML Pointing Language)** permettent de définir des mécanismes de liens entre différents types de ressources. XLink permet de définir des liens entre différents documents ou parties de documents, grâce à un mécanisme plus puissant que celui de HTML. XLink permet de définir des liens simples (comme en HTML), étendus (qui associent plusieurs ressources) ou des arcs (qui définissent les règles de passage entre les ressources). XML Base permet de définir un mécanisme d'URI (Uniform Resource Identifier, voir plus loin) de base vers des documents XML, similaire à celui utilisé en HTML. XPointer permet de faire pointer des liens vers des sous-parties de documents. La syntaxe de XPointer pour XML est la même que celle utilisée en HTML : le nom du fichier est suivi du caractère '#' (dièse) et de l'identifiant de la ressource vers laquelle pointe le lien. Ces langages ne sont pas détaillés dans ce cours.

Le langage de feuilles de styles associé à XML, *XSL* (Extended StyleSheet Language) permet la mise en forme de documents XML. Il se compose des langages *XPATH*, qui permet la localisation des éléments et des parties dans un document XML, et *XSLT* (XSL Transformations), qui définit le format de sortie du document. L'association d'XSL à un document XML permet de compléter le couple données / méta-données défini dans le document en lui associant des informations de formatage. Il existe aussi un troisième langage, *XSL-Formatting Objects*, qui permet d'aller plus loin dans la composition documentaire, en décrivant notamment la structure physique des pages des documents générés. XSL-FO n'est pas détaillé dans ce cours.

Les information sets XML : il est courant de trouver, dans les spécifications d'applications et d'outils pour XML, le terme « *XML-infoset* ». Ce terme fait référence à un ensemble

[18] Le DOM HTML est utilisé pour accéder (et éventuellement modifier) dynamiquement les éléments d'un document HTML. L'implémentation du DOM est cependant très dépendante des navigateurs (ex : *document.balise* sous Netscape™ ; *document.all.balise* sous IE).

d'informations abstrait, ou jeu de données abstrait, permettant de considérer une partie des donnée d'un document XML indépendamment de ce document et des outils logiciels utilisés pour le traiter (comme le DOM ou SAX, voir plus loin). La seconde version de la notion d'information set date du 4 février 2004 et fait suite à la version 1.1. de XML.
La notion d'infoset ne représente rien de plus qu'une partie d'un arbre XML. Les données contenues dans cet arbre sont appelées information items et sont de plusieurs types : il existe 11 types d'information items, parmi lesquels : document (le document XML entier), *element* (un élément du document, au sens de la recommandation XML), *attribute* (un attribut d'un élément XML), *processing instruction* (instruction de traitement), *character* (données de type CDATA) ou *unparsed entity* (référence à une entité XML non analysée).
Chacun de ces types d'information item possède un ensemble spécifique de *propriétés*, qui permettent d'identifier les données du document, leur position dans l'arbre ou toute autre caractéristique utile en fonction du type d'item.

2.4.4 Description d'un document XML

Dans les paragraphes précédents, vous avez eu, par quelques exemples, un aperçu de parties de documents XML. Le paragraphe suivant détaille les différents éléments de structures d'un document XML.

2.4.4.1 Déclaration et instructions de traitement

Déclaration simple d'un document XML : `<?xml version="1.0"?>` Une déclaration XML n'est *a priori* pas obligatoire, le serveur étant capable d'indiquer qu'il envoie du XML. Cependant, il est recommandé de l'inclure, notamment pour y faire figurer le numéro de version.

Remarque : si cette déclaration est incluse, l'attribut « `version` » doit nécessairement y figurer.

Encodage : `<?xml version="1.0" encoding="UTF-8"?>`. UTF-8 et UTF-16 sont les formats ASCII qui sont gérés en interne par tous les processeurs XML. Ces encodages sont détectés par défaut et n'ont donc pas besoin d'être spécifiés. En revanche, la présence de caractères incompatibles avec l'un de ces deux formats sans précision du type d'encodage utilisé entraîne une erreur du processeur.

Remarque : d'autres encodages peuvent être utilisés pour inclure des caractères n'existant pas dans les jeux américains UTF-8 et UTF-16. En particulier, les jeux de caractères adaptés au français (accents, cédilles…) dans des documents XML sont « `windows-1252` », qui n'est pas accepté par tous les parsers, ou « `ISO-8859-1` », qui lui est très semblable et plus souvent reconnu. Certains éditeurs permettent également d'enregistrer du texte directement en unicode, qui est le standard international, mais peu de parseurs permettent à l'heure actuelle l'interprétation de ce jeu de caractères. Cela devrait changer avec le passage à XML 1.1.

SDD (Standalone Document Declaration) : dans la déclaration `<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>`, le dernier attribut (standalone) permet de déterminer si le document est *autonome* (on trouve aussi *autodescriptif*). Si ce n'est pas le cas, c'est qu'il fait référence à une DTD (ou un schéma XML) externe. Par défaut, un document XML est considéré comme autonome, puisque la DTD peut être omise.

Remarque : les attributs version, encoding et standalone qui figurent dans la déclaration doivent y figurer dans cet ordre.

Instructions de traitement : d'une façon générale, les lignes encadrées par "<?" et "?>" sont appelées *instructions de traitement* (Processing Instructions, ou *PI*). Ces instructions n'ont aucune signification pour le document XML, et sont transmises directement à l'application par le parser pour être traitées par elle. La syntaxe générale d'une PI est : `<?Application_destinatrice Texte_de_l'instruction(avec ou sans espaces)?>`.

Remarque : officiellement, la déclaration XML n'est pas une instruction de traitement, car elle ne concerne pas l'application qui utilise le document XML, mais est uniquement destinée au parser. Un parser n'est donc pas obligé de la transmettre à l'application. En pratique, cette distinction n'a aucune incidence sur ce cours. Nous considérons ici la déclaration XML comme une instruction de traitement, puisqu'elle respecte la syntaxe de ces instructions.

2.4.4.2 Document bien formé

Un document est dit bien formé s'il respecte la syntaxe XML. Il n'a pas besoin d'être conforme à un quelconque modèle de document.

Un document XML est composé d'un *prologue* et d'un *élément* (appelé élément racine).

Un prologue est composé éventuellement d'une déclaration XML et éventuellement d'une *déclaration de type de document*.

Un élément est composé soit d'une *balise ouvrante*, éventuellement d'un *contenu* et d'une *balise fermante*, soit d'une balise d'*élément vide*.

Une balise ouvrante est entourée de chevrons et contient un nom d'élément et éventuellement des attributs.

Une balise fermante est entourée de chevrons, contient un slash ('`/`') et un nom d'élément.

Une balise d'élément vide est entourée de chevrons, contient un nom d'élément, éventuellement des attributs et un slash.

Un contenu est composé de texte et / ou d'autres éléments appelés fils de l'élément courant. Le texte est appelé « données caractères analysables » ou PCDATA (pour Parsed Character DATA). Le contenu autorisé pour le PCDATA dépend du type d'encodage choisi.

Remarques :

- Le fait d'avoir un unique élément racine, décomposable en éléments fils, eux-mêmes décomposables, etc. définit une structure arborescente, assimilée à un modèle objet du document.
- L'inclusion du contenu entre les balises d'ouverture et de fermeture de l'élément ne permet pas le chevauchement de balises entre un élément père et un élément fils.
- Dans tous les cas, les caractères "<" (inférieur) et "&" (esperluète) sont interdits dans les contenus. On aura recours aux *entités* "`<`" et "`&`".

- L’usage de ">" (supérieur) ou des guillemets simples ou doubles peut également être perturbant. Dans ce cas, on a recours à &gt ;, &apos ; et &quot ;.
- Si l’on veut vraiment utiliser les caractères "<" ou "&", il est possible de définir une balise sous forme de zone de caractères non analysés, sous la forme : `<![CDATA [ texte comprenant des caractères interdits ]]>`. Ici, CDATA s’oppose à P(arsed)CDATA, qui autorise l’analyse des données textuelles.
- Le mode de traitement des espaces peut être défini grâce à l’attribut *xml-space*, qui peut prendre les valeurs *default* ou *preserve*.
- Par défaut, les éléments définis comme de type PCDATA conservent les espaces. Le mode de traitement des espaces pour les autres éléments dépend de l’application.

Un nom d’élément en XML 1.0 peut contenir des lettres, des chiffres, des modificateurs de lettres (si cela est autorisé par l’attribut « encoding » de la déclaration XML, et les caractères "`.`" (point), "`-`" (tiret), "_" (souligné) ou "`:`" (deux-points). En XML 1.1, cette liste reste valable, mais elle est augmentée de tous les caractères Unicode qui ne sont pas spécifiquement interdits par la recommandation (http://www.w3.org/TR/2004/REC-xml11-20040204).

Remarques :
- Un nom doit commencer par une lettre ou par le caractère "_" et non par un chiffre ou un autre signe de ponctuation.
- Un nom ne peut pas commencer par la séquence de lettres x, m, l, quelle qu’en soit la casse.
- XML est sensible à la casse des noms d’éléments.
- Éviter les deux points dans le nom des éléments, car ils sont utilisés pour spécifier les espaces de noms (voir plus loin).
- **Les attributs** possèdent un nom unique pour chaque élément et une valeur. La syntaxe employée en XML est tout d’abord le nom de l’attribut, puis le caractère "=" (égal), puis sa valeur entre guillemets simples ou doubles.

Les commentaires débutent par la chaîne « `<!--` » et se terminent par la chaîne « `-->` ». Ils ne peuvent être placés à l’intérieur d’une balise et ne sont pas obligatoirement transmis à l’application.

2.5 Document XML valide

L'opération de validation est réalisée par le parser après l'analyse de la syntaxe (i.e. uniquement si le document est bien formé). Pour cela, il faut que le parser utilisé soit validant (et éventuellement que l'option de validation soit activée[19]). La validation d'un document peut se faire d'après une DTD ou d'après un schéma XML.

2.5.1 Les DTD

Une DTD est une *définition de type de document*. On s'y réfère pour spécifier le modèle de données auquel appartient un document. Ce type de définition est hérité de SGML et permet de valider la structure du document. La spécification des DTD XML fait partie intégrante de la recommandation XML 1.0. Pour qu'un document soit validé par une DTD, il faut :

- qu'il contienne une (et une seule) *déclaration de type de document*,
- qu'il fasse référence, de façon interne ou externe, à une DTD décrivant sa structure.

Remarque : si une DTD décrit la structure du document, la valeur du Standalone Document Declaration de la déclaration XML (voir page 50) est forcément « `standalone="no"` ».

La déclaration de type de document est de la forme :

`<!DOCTYPE Nom_de_l_élément_racine Type_de_source emplacement1 emplacement2 [sous-ensemble interne de DTD]>`, où :

`Nom_de_l_element_racine` représente – contre toute attente – le nom de l'élément racine du document XML. En conséquence, il ne peut y avoir de DTD pour une partie de l'arborescence du document.

Le type de source est associé aux mots-clé « `SYSTEM` » ou « `PUBLIC` ».

- `SYSTEM` correspond à une DTD employée ponctuellement pour décrire le document. Elle est associée à un emplacement indiqué sous forme d'une URI. Le mot-clé `SYSTEM` permet de mettre la DTD en cache et de la rendre disponible hors connexion.
- Une DTD déclarée `PUBLIC` est une DTD qui peut être partagée. Elle a donc une portée plus large qu'une DTD `SYSTEM`. Dans le cas de deux emplacements, le second est un « emplacement de secours », pour le cas où le premier est inaccessible. Pour cela, le premier emplacement indiqué est un URI, qui désigne pour le processeur l'emplacement local de la DTD.

`[sous-ensemble interne de DTD]` désigne une éventuelle partie de la définition de la structure du document incluse dans celui-ci. En l'absence de DTD externe, il est utilisé pour définir l'ensemble de la structure du document. S'il existe une DTD externe, il est essentiellement utilisé pour déclarer des entités ou attributs (voir plus loin) spécifiques au document. Dans ce dernier cas, le sous-ensemble interne est utilisé pour définir des caractéristiques sans effet sur les autres documents faisant référence à la DTD externe.

---

[19] Par exemple, MSXML n'effectue pas de validation de documents XML, lorsque ceux-ci sont affichés avec la feuille de style par défaut. Il existe un « flag » appelé validateOnParse qu'il faut positionner (via le DOM) pour forcer la validation.

La définition de type de document est dérivée de SGML. Elle permet de décrire la structure du document dans un formalisme spécifique. Une DTD peut comporter un sous-ensemble externe et un sous-ensemble interne. Dans ce cas, c'est le sous-ensemble interne qui a priorité sur le sous-ensemble externe. Une seule DTD est toutefois autorisée par document.

Une DTD décrit quatre types d'éléments :
- les éléments du document XML, de façon arborescente, en partant de l'élément racine (mot-clé : `ELEMENT`),
- les attributs des différents éléments, sous forme de liste (mot-clé : `ATTLIST`).
- les différentes entités auxquelles font référence les éléments du document (mot-clé : `ENTITY`),
- les éventuelles notations, qui déclarent du contenu non XML, comme des données graphiques ou binaires (mot-clé : `NOTATION`),

Un élément est déclaré comme suit : `<!ELEMENT nom_element (contenu)>`.

Il peut avoir cinq types de contenu :
- autre(s) élément(s) (il est composé uniquement d'éléments fils),
- `PCDATA` (pour « Parsed Character DATA » : du texte analysable conforme au jeu de caractères spécifié dans la déclaration XML),
- mixte (élément(s) + `PCDATA`),
- `ANY` (n'importe quel type de contenu XML bien formé),
- `EMPTY` (vide…).

Dans les cas où il contient d'autres éléments ou du contenu mixte, ce contenu peut être décrit sous forme de liste entre parenthèses `"("` `")"`, et séparées par une virgule `","` (pour signifier un ET logique entre ces éléments) ou un pipe `"|"`(pour un OU logique). Il est possible d'imbriquer les parenthèses pour définir des priorités entre les opérateurs logiques. Dans le cas du ET, il est impératif de respecter l'ordre des éléments dans la DTD. Dans le cas de contenus mixtes, `PCDATA` apparaît toujours en début de liste dans la DTD, même si dans le document, les données textuelles peuvent apparaître n'importe où dans l'élément.

Par ailleurs, il existe des *opérateurs de cardinalité* qui spécifient le nombre d'occurrences de chaque type d'élément enfant compris dans le contenu de l'élément :
- Aucun opérateur signifie que le contenu apparaît une fois et une seule.
- `?` indique 0 ou une occurrence du contenu (contenu facultatif).
- `*` indique 0, une ou plusieurs occurrences du contenu (contenu facultatif)
- `+` indique une ou plusieurs occurrences (contenu obligatoire).

Les attributs sont déclarés tous ensemble dans la liste d'attributs de chaque élément. La syntaxe de cette déclaration est la suivante :

```
<!ATTLIST nom_element
      nom_attribut type_attribut déclaration_valeur_implicite
      ...
      nom_attribut type_attribut déclaration_valeur_implicite>
```

Les types d'attributs autorisés sont les suivants :
- `CDATA` : chaîne textuelle simple (non analysable, par opposition à `PCDATA`).

- Valeurs énumérées dans une liste de choix entre parenthèses et séparés par des pipes.
- `ID` : un identifiant unique respectant la syntaxe des noms d'éléments XML. Un seul attribut `ID` est autorisé par élément.
- `IDREF` : référence à un élément ayant pour valeur de l'attribut `ID` la valeur indiquée.
- `IDREFS` : liste de plusieurs IDREF séparés par des espaces.
- `NMTOKEN` : jeton de nom conforme aux règles de noms XML (permet de limiter le nombre de valeurs que peut prendre l'attribut).
- `NMTOKENS` : liste de jetons de noms séparés par des espaces.
- `ENTITY` : nom d'une entité prédéfinie.
- `ENTITIES` : liste de plusieurs entités séparés par des espaces.
- `NOTATION` : type de notation déclaré ailleurs dans la DTD.

La déclaration de valeur implicite d'un attribut peut avoir les valeurs « `#REQUIRED` » (obligatoire), « `#IMPLIED` » (facultatif), « `#FIXED` » + une valeur entre double guillemets (s'il est présent, il doit obligatoirement avoir cette valeur) ou une valeur par défaut entre double guillemets (qui peut utiliser des caractères génériques).

Une entité est une séquence de caractères repérée par un nom. Ce nom peut ensuite être référencé dans un document XML ou dans la DTD et le parser transforme cette référence en la valeur de l'entité. Celle-ci peut être de deux types : générale ou paramètre.

Il existe deux types d'entités générales : analysables (internes ou externes à la DTD), ou non analysables (toujours situées dans un fichier externe à la DTD). Les entités analysables internes sont définies sous la forme `<!ENTITY nom "texte_de_remplacement" >`, où texte de remplacement est un texte bien formé quelconque, ne contenant pas de référence directe ou indirecte à cette entité.

Une entité générale non analysable se trouve toujours dans un fichier ou une ressource externe. Elle est définie par forme `<!ENTITY nom SYSTEM "localisation" NDATA type_de_notation>` ou `<!ENTITY nom PUBLIC "localisation1" "localisation2" NDATA type_de_notation>`. Chaque NDATA doit correspondre au nom d'une déclaration de notation pour être valide.

Les entités paramètres sont uniquement utilisées dans les DTD et doivent toujours être analysables. Elles sont définies sous la forme :
`<!ENTITY % nom "texte_de_remplacement" >`.

Les références d'entités se font par &nom; pour les entités générales et %nom; pour les entités paramètres.

Les notations permettent de déclarer du contenu externe non XML ainsi que les applications à utiliser pour les traiter. Elles sont définies comme les entités externes, sous la forme :
`<!NOTATION nom SYSTEM "localisation" >` ou
`<!NOTATION nom PUBLIC "localisation1" "localisation2" >`.

2.5.2 Les schémas XML

Bien que les DTD soient la méthode de validation de documents prescrite par la recommandation XML 1.0, la description de documents XML par les DTD compte un certain nombre de limites. Une alternative aux DTD est proposée par le concept de schémas XML. Le document de travail concernant les schémas XML date du 10 avril 2000 ; il est devenu une recommandation officielle du W3C depuis le 2 mai 2001. Cette recommandation est disponible en trois parties. La seconde édition date du 28 octobre 2004.

XML Schema part 0: Primer (introduction) : http://www.w3.org/TR/xmlschema-0

XML Schema part 1: Structures : http://www.w3.org/TR/xmlschema-1

XML Schema part 2: Datatypes : http://www.w3.org/TR/xmlschema-2

Ce cours résume les principales caractéristiques de schémas XML tels qu'ils ont été définis dans cette recommandation. Il s'est également inspiré d'ouvrages traitant de ces schémas publiés avant la publication définitive de la recommandation et comportant une syntaxe parfois différente. Le maximum a été fait pour corriger cette variabilité et rendre les éléments de syntaxe présentés dans ce cours conformes à la recommandation. En tout état de cause, les concepts fondamentaux liés aux schémas sont identiques dans la recommandation et dans tous les ouvrages. Le lecteur est simplement invité à s'assurer de la conformité de la syntaxe présentée ici avec celle de la version des schémas qu'il utilise.

2.5.2.1 Comparaison schémas et DTD

Le tableau suivant récapitule les principales différences entre les schémas XML et les DTD. Il permet de se rendre compte des différences existant entre ces deux concepts.

Caractéristique	DTD	Schémas
Syntaxe	Notation EBNF (Extended Backus-Naur Form) + pseudo-XML	XML 1.0
Outils	Outils SGML existants (chers et complexes)	Tous les outils XML existants et à venir (DOM, XSLT, navigateurs)
Supports DOM/SAX	Non	Oui (affichage et manipulation comme pour les fichiers XML).
Modèles de contenu	- Listes : ordonnées ou de choix - Cardinalité : 0, 1 ou plusieurs occurrences - Pas d'éléments nommés ou de groupes d'attributs.	- Listes : ordonnées et de choix (détails de contenus mixtes) - cardinalité : spécification d'un nombre exact d'occurrences possible - groupes de modèles nommés
Typage des données	Faible (chaînes, jetons nominaux, ID…)	Fort (nombres, chaînes, date/heure, booléen, structures…)
Portée des noms	Globale	Globale ou locale
Héritage	Non	Oui
Extensibilité	Non (pas sans modification de la recommandation XML 1.0)	Oui (puisque fondés sur l'extensibilité de XML)
Contraintes légales	Compatibilité avec SGML	Aucune (simplement des « emprunts » aux DTD, comme pour les types de données)
Nombre de vocabulaires supportés	Une seule DTD par document	Autant que nécessaire (grâce aux espaces de noms)
Dynamicité	Aucune (les DTD sont en lecture seule)	Peuvent être modifiés en cours d'exécution (par exemple avec le DOM)

2.5.2.2 Caractéristiques des schémas XML

Syntaxe : contrairement aux DTD, dont la syntaxe particulière est héritée des spécifications de SGML, les schémas XML sont des parties de document XML. Ils respectent donc la syntaxe du langage. Cela leur confère la même extensibilité que XML, et permet de les manipuler avec les mêmes outils. En particulier, on peut utiliser le DOM ou SAX pour analyser et même modifier dynamiquement un schéma XML en cours d'utilisation.

Modèles de contenu : Comme nous l'avons vu, les DTD ne permettent pas de spécifier des modèles de contenus précis. Tout au plus, peut ont indiquer une liste des différents types d'éléments et des indications de cardinalité peu précises. Les schémas XML permettent en revanche de spécifier avec tout le niveau de détail nécessaire les modèles de contenu. Les contenus mixtes peuvent alors être décrits et validés précisément, aussi bien pour les données XML traditionnelles (i.e. « documentaires ») que pour les applications XML récentes ou futures.

Typage des données : pour permettre la compatibilité avec toutes ces applications, les schémas sont capables de prendre en charge tous les types de données. Cela est réalisé par deux mécanismes distincts, qui font l'objet de deux documents distincts des spécifications du W3C : les *types de données* (datatypes) et les *structures*. Nous revenons sur ces mécanismes plus loin.

Extensibilité : alors que la syntaxe des DTD est figée (tout ajout à cette syntaxe nécessite une modification de la recommandation XML 1.0), l'extensibilité des schémas XML est principalement due à l'absence de limitation dans l'utilisation des types de données. De plus, elle profite du partage de vocabulaires, grâce à la prise en charge des espaces de noms XML.

Dynamicité : Outre la modification dynamique d'un schéma à l'aide du DOM (peu recommandée, car la capacité de prise en charge de ce type de modifications est très dépendante du processeur utilisé), il est possible de sélectionner dynamiquement (i.e. en fonction des actions de l'utilisateur), le schéma ou la partie de schéma à appliquer à un élément d'un document XML.

Par exemple, après avoir effectué une recherche bibliographique en ligne à l'aide d'un schéma approprié, un utilisateur peut décider de commander un livre, à l'aide d'un schéma totalement différent et choisi dynamiquement en fonction du pays à partir duquel il réalise sa commande.

2.5.2.3 Principes de base des schémas

Le but d'un schéma est de définir une « classe de documents XML », également souvent désignée sous le terme *instance de document*. Ni les instances ni les schémas n'existent indépendamment les uns des autres. De plus, ces termes ne font pas directement référence à des fichiers à proprement parler. Il s'agit dans les deux cas de parties de documents XML, qui peuvent cependant constituer des fichiers entiers.

Nous avons vu que la spécification officielle du W3C propose deux mécanismes de définition des données : les types de données et les structures. Il existe également deux modes de balisage, qui correspondent – à peu près – à ces deux parties de la spécification : les *définitions* et les *déclarations*.

Bien que le W3C ait choisi de spécifier d'abord les structures et ensuite les types de données (après les avoir cependant présentés dans l'ordre inverse dans le document non normatif d'introduction), nous faisons ci-dessous l'inverse.

2.5.2.4 Les types de données

Ce paragraphe décrit les différentes caractéristiques des types de données utilisés dans les schémas XML. Il présente les trois dichotomies utilisées pour décrire les types de données, en fonction du fait qu'ils sont *primitifs* ou *dérivés*, *intégrés* ou *dérivés par l'utilisateur* et définis de façon *atomique* ou par *listes*.

Comme dans la plupart des langages de programmation « évolués », tous les types de données des schémas sont définis de façon arborescente dans une *hiérarchie des définitions de type*. La racine conceptuelle de cette arborescence se nomme *définition d'ur-type*. La distinction entre les types de données primitifs et dérivés permet de les situer dans cette hiérarchie. Les types primitifs sont alors les premiers descendants de l'ur-type. Les types dérivés constituent tous les niveaux inférieurs de cette décomposition.

La deuxième classification distingue les types de données déterminés par les spécifications des schémas XML des types définis dans le corps des schémas. Les premiers sont les types intégrés (« built-in »). Ils sont automatiquement reconnus par le processeur XML et peuvent être utilisés dans tout schéma fondé sur la recommandation du W3C. La définition des types dérivés par l'utilisateur fait appel à la notion de structure qui est présentée plus loin.

Remarque : la relation entre ces deux premières classifications est simple : TOUS LES TYPES PRIMITIFS SONT INTÉGRÉS. La réciproque n'est cependant pas vraie : il existe des types intégrés dérivés. Comme leur nom l'indique, tous les types de données dérivés par l'utilisateur sont dérivés.

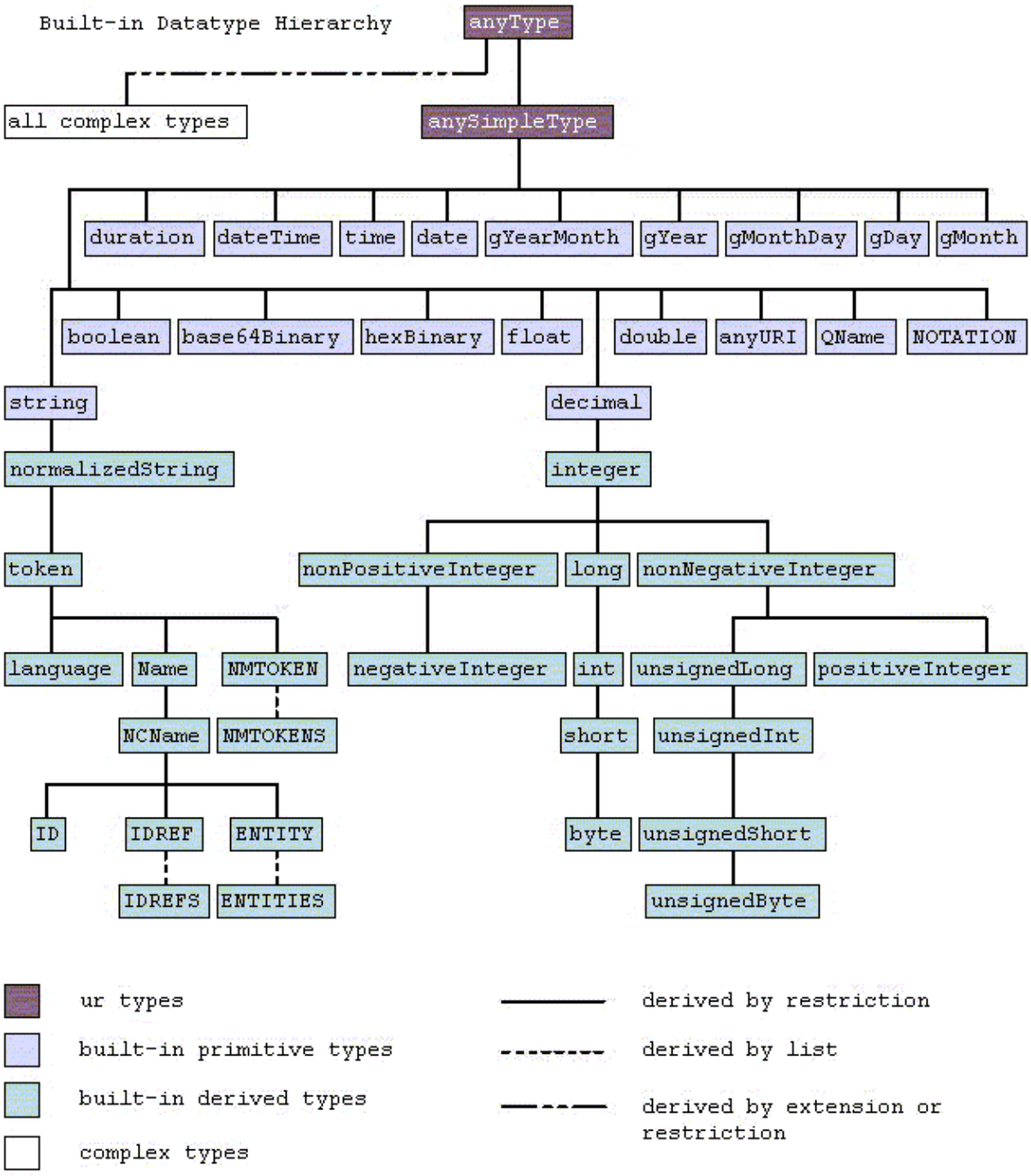
La troisième distinction sépare les types de données atomiques et listes. Les premiers sont constitués de valeurs « indivisibles » (au sens des schémas XML). Les seconds sont constitués d'une séquence délimitée de valeurs atomiques.

Remarque : la relation entre types atomiques / listes et types primitifs / dérivés n'est pas directe. LES TYPES ATOMIQUES NE SONT PAS DES TYPES PRIMITIFS. Il existe des types atomiques primitifs et dérivés. En revanche, les types listes sont une catégorie de types dérivés.

Les types de données primitifs peuvent être considérés comme la matière première de définition des éléments XML. Ils peuvent être utilisés pour spécifier les *valeurs* d'éléments ou d'attributs, mais ne peuvent pas avoir de *contenu* (i. e. des éléments enfants ou des attributs).

Les types de données dérivés sont définis à partir d'un type de données existant (le type de base). Ils peuvent dériver d'un type primitif ou d'un autre type dérivé. Un type de données dérivé peut contenir un texte XML bien formé et valide (selon la définition du type de données de base), ainsi que des attributs.

La figure ci-dessous (extraite de la recommandation du W3C) donne la liste de tous les types de données intégrés.



Les types de données atomiques peuvent être primitifs (comme les chaînes de caractères, puisque le type caractère n'existe pas) ou dérivés (comme les entiers, qui dérivent du type float). Les exemples suivants présentent respectivement des éléments de ces deux types atomiques.

```
<atome_primitif>Cette chaine est appropriee parce qu'elle ne peut etre divisee en caracteres</atome_primitif>
<atome_derive>231344</atome_derive>
```

Les types de données listes sont toujours des dérivés de types atomiques et ne peuvent être constitués d'autres listes. Les éléments d'une liste sont toujours délimités par des espaces. De ce fait, les espaces ne peuvent intervenir à l'intérieur de l'une des valeurs de la liste (comme c'est par exemple le cas dans une chaîne de caractères). Par exemple, si l'on définit une liste

`taille`, ayant pour base le type intégré `decimal` (la syntaxe d'une telle définition est présentée plus loin), un élément qui utilise ce type de liste est :
`<Pointures xsi:type="tailles">8 8.5 9 9.5 10 10.5 11 12 13</Pointures>`

Remarque : de la même façon, une liste ayant pour base le type atomique `string` et présentée sous la forme : `<ListeTexte>Element un, element deux, element cent-quatre-vingt-douze</ListeTexte>` comporte cinq espaces et donc six éléments.

2.5.2.5 Les facettes des types de données

Tous les types de données possèdent trois caractéristiques : un *espace de valeurs*, un *espace lexical* et des *facettes*.

L'espace de valeurs d'un type de données désigne l'ensemble des valeurs pouvant être prises par les éléments qui s'y réfèrent. Ces espaces de valeurs sont implicites dans la définition des types primitifs. Par exemple, l'espace de valeurs lié au type `float` va de moins l'infini à plus l'infini. Les types de données dérivés héritent des espaces de valeurs de leurs types de base. Par conséquent, l'espace de valeurs du type `integer` est le même que celui du type `float`.

L'espace lexical d'un type de données désigne la chaîne de caractères *représentant* la valeur de l'élément. Ces chaînes comportent toujours du texte valide en XML, en fonction de l'encodage choisi. Les chaînes de caractères n'ont par exemple qu'une seule représentation lexicale, alors que d'autre types peuvent en avoir plusieurs. Par exemple, dans l'espace de valeurs «nombres à virgule flottante », 100, 1^E2 et 10^2 représentent le même nombre. Ces représentations sont cependant différentes au niveau lexical.

Les facettes désignent les propriétés définitionnelles des types de données, permettant de les différencier les uns des autres. Il existe des propriétés abstraites, nommées *facettes fondamentales*, et des limites facultatives sur l'espace de valeurs des types de données, appelées *facettes de contrainte*.

Il existe cinq **facettes fondamentales** :

- *égalité* : cette propriété s'applique à tous les types de données, numériques ou non, et peut dépendre de la casse et de l'encodage des valeurs,
- *ordre* : les relations d'ordre sont intrinsèques pour les valeurs numériques et textuelles (pour ces dernières, elles dépendent également de l'encodage utilisé),
- *bornes* : les valeurs d'un type de données peuvent avoir une borne supérieure ou inférieure ou les deux ; dans ce dernier cas, l'espace de valeurs est considéré comme borné,
- *cardinalité* : tous les types de données sont concernés par cette propriété ; la cardinalité d'un espace de valeurs peut être finie (comme dans le cas d'une liste de valeurs énumérées), infinie dénombrable ou infinie indénombrable,
- *numérique / non numérique* : cette propriété est vraie si les valeurs prises par le type de données appartiennent à un système de nombres.

Il existe 12 **facettes de contraintes** :

- `length`, `minlength`, `maxlength` : traitent le nombre d'unités de longueur d'un type de données en nombres de « points de code Unicode » (i.e. sur 8, 16 ou 32 bits) pour les chaînes de caractères, en nombres d'octets pour les données de types binaires et dérivés et en nombres d'éléments pour les données de types listes,
- `pattern` : contrainte de l'espace lexical, qui limite l'espace des valeurs, via une expression régulière « regex » (langage proche de PERL),
- `enumeration` : limite l'espace de valeurs à une liste de choix énumérées,
- `whiteSpace` : s'applique uniquement aux types dérivés de string ; contraint la manière dont sont traités les caractères d'espacement ; cette facette peut prendre les valeurs `preserve` (aucune modification n'est apportée à la mise en forme), `replace` (les caractères de tabulation et de retour à la ligne sont remplacés par des espaces) ou `collapse` (effet identique à la précédente, plus remplacement des séquences de caractères d'espacement par un caractère d'espacement unique),
- `minInclusive`, `maxInclusive`, `minExclusive`, `maxEclusive` : définissent les bornes inférieures et supérieures de l'espace de valeurs, par des inégalités larges ou strictes,
- `totalDigits` : s'applique uniquement aux données de type `decimal` (ou dérivées de ce type) ; cette facette définit le nombre total de chiffres que peuvent prendre les données,
- `fractionDigits` : s'applique uniquement aux données de type `decimal` (ou dérivées de ce type); cette facette définit le nombre de chiffres de la partie décimale des données.

2.5.2.6 Les structures

Les structures permettent de décrire des types d'éléments, des attributs d'éléments et des *modèles de contenu* d'éléments. Pour cela, elles s'appuient sur la définition de types de données, selon les mécanismes de dérivation évoqués plus haut. Il existe des types de données *simples* et *complexes*. Toutes les structures de données présentées ici correspondent donc à des types de données dérivés par l'utilisateur[20].

2.5.2.6.1 Les types simples

Ils consistent à dériver un type de données atomique[21] par restriction, par liste ou par union. La dérivation par restriction s'obtient en spécifiant un ensemble de contraintes sur les espaces de valeurs ou lexical d'un type de base. La dérivation par liste est la spécification de l'ensemble des valeurs pouvant être prises par un élément du type concerné. La dérivation par union permet de définir un type simple de données en tant que sur-ensemble d'autres types simples.

---

[20] D'après les spécifications du W3C, la plupart des éléments XML décrits dans cette partie peuvent contenir un élément `<annotation>`. Les annotations sont utilisées pour transmettre des commentaires soit aux lecteurs humains du schéma, soit à l'application. Pour cela, elles contiennent respectivement des éléments enfants `<documentation>` et `<appInfo>`. Elles ne sont pas plus détaillées ici. L'inclusion de cette balise dans les éléments de définition de types est quasi-systématique et n'est plus mentionnée dans la suite.

[21] La dérivation de types listes n'est pas autorisée.

Remarque : les modes de dérivation par restriction et par liste correspondent au concept de définition par intension ou par extension. Dans le premier cas, on définit un ensemble par les propriétés de ses éléments. Dans le second, on nomme explicitement tous ses éléments.

La définition de types simples est encadrée par l'élément `<simpleType>`. Comme les types primitifs dont ils héritent de plus ou moins loin, les types simples ne peuvent avoir de contenu. Il ne sont utilisables que pour spécifier des valeurs d'attributs ou de contenus d'éléments n'ayant pas d'enfants.

Attributs : l'élément `<simpleType>` prend les attributs suivants.

- `name` : NCName (nom du type de données) ; facultatif (dans ce cas, le type est dit *anonyme* et s'applique à l'élément, attribut ou annotation de niveau immédiatement supérieur),
- `final` : `#all`, ou (`list` ou `union` ou `restriction`) ; spécifie les modes de dérivation pour lesquels le type ne peut pas être dérivé ; facultatif (valeur par défaut : aucun mode de dérivation),

Contenu : le mode de dérivation est indiqué par un élément enfant de l'élément `<simpleType>` parmi l'un des trois suivants : `<restriction>`, `<list>` ou `<union>`.

`restriction` doit posséder un attribut `base` (Qname) qui a pour valeur le nom du type de données de base ; cet attribut est facultatif (valeur par défaut : ur-type). Le contenu des descriptions de types simples dérivés par restriction est constitué d'une ou de plusieurs facettes de contrainte représentées par des éléments enfants vides. Exemple :

```
<simpleType name=”EntierNegatif”>
  <restriction base=”xsi:integer”>
    <minInclusive value="unbounded" />
    <maxInclusive value="-1" />
  </restriction>
</simpleType>
```

`list` peut avoir l'attribut `itemType` qui indique le type d'items auquel appartiennent les éléments dont on va donner la liste ou un élément enfant `<simpleType>` qui définit ce type de façon anonyme ; `itemType` et `<simpleType>` sont exclusifs dans un élément `list`, mais l'un des deux est requis, comme dans l'un des deux exemples suivants ;

```
<simpleType name="tailles">
  <list itemType="decimal"/>
</simpleType>
```

```
<simpleType name="tailles">
  <list>
    <simpleType>
      ...
    </simpleType>
  </list>
</simpleType>
```

Cette définition de type permet alors l'emploi dans le schéma du type liste `tailles`, comme dans l'exemple `Pointure` présenté précédemment.

Remarque : de la même façon que pour la dérivation de types par restriction, il est possible de faire figurer des facettes de contraintes ou des annotations dans une définition de liste.

union peut avoir l'attribut memberTypes qui indique les types d'items auxquels peuvent appartenir les éléments ou un ou plusieurs éléments enfants <simpleType> anonymes. memberTypes et <simpleType> ne sont pas exclusifs dans l'élément union (contrairement à l'élément list) :

```
<simpleType name="tailles">
  <union memberTypes="xsi:integer">
    <simpleType>
      <restriction base="decimal">
        <fractionDigits value="1"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Remarque : cette spécification de la syntaxe de définition de l'élément <simpleType> est extraite de la recommandation du W3C sur les schémas datant du 02 mai 2001. Elle diffère des versions précédentes en plusieurs points. En particulier, l'attribut derivedBy disparaît, puisque le mode de dérivation est indiqué par un élément enfant. D'autre part, il n'est plus possible de définir des types simples abstraits, comme cela était autorisé dans des versions précédentes.

2.5.2.6.2 Les types complexes

Ils permettent de spécifier des gammes de contenus plus vastes que ceux des types simples. Ils sont créés par la dérivation d'autres types. Une définition de type complexe est :

- soit une dérivation la restriction d'un type de base complexe,
- soit une dérivation par extension d'un type de base (simple ou complexe),
- soit une dérivation par restriction de l'ur-type definition.

Les types complexes définissent :

- des attributs appartenant aux attributs du type de base,
- des contenus complexes, conformes à un *modèle de contenu* dérivé de celui du type de base.

Par rapport aux types simples, les types complexes proposent en plus une syntaxe de **description des contenus**. Ils sont introduits par l'élément <complexType> et possèdent les attributs suivants :

- *name* : NCName (nom du type de données) ; facultatif (valeur par défaut : l'élément de niveau immédiatement supérieur),
- *mixed* : booléen (l'utilisation de cet attribut n'est pas autorisée si le contenu est de type simple) ; facultatif (valeur par défaut : false)
- *abstract* : booléen (si true, le type ne peut pas être utilisé pour la validation d'un contenu ou d'un attribut ni être dérivé ; ils peuvent juste être utilisés en tant que types de base ou pour spécifier le type d'un élément dans sa déclaration) ; facultatif (valeur par défaut : false),
- *final* : « #all », « extension », « restriction » ou vide (indique la *finalité* du type de données pour chaque mode de dérivation ; un type d'élément final pour un mode de dérivation ne peut pas être dérivé suivant ce mode) ; facultatif

(valeur par défaut : vide ; tous les modes de dérivation sont autorisés) ; cet attribut n'est pas hérité par les types dérivés,

- *block* : « `#all` », « `extension` », « `restriction` » ou vide (permet de spécifier les modes de dérivations qui ne doivent pas être utilisés) ; facultatif (valeur par défaut : la valeur de l'attribut `blockDefault` de l'élément racine `<schema>` s'il y en a une ; sinon : vide) ; cet attribut est hérité par dérivation.

Les définitions de types complexes permettent de spécifier des modèles de contenus constitués d'attributs et de contenus.

La spécification d'attributs est faite grâce à l'élément `<attribute>` qui peut contenir des annotations ou un élément `<simpleType>`. Cependant, en règle générale et à l'exception des annotations, cet élément est souvent vide. Il possède les attributs suivants :

- `default` : string (valeur par défaut de l'attribut) ; incompatible avec `fixed`,
- `fixed` : string (valeur constante de l'attribut) ; incompatible avec `default`,
- `form` : « `qualified` » ou « `unqualified` » (définit si l'attribut appartient à un espace de noms ; le cas échéant, il doit être préfixé par le préfixe correspondant à l'espace de noms de niveau immédiatement supérieur) ; incompatible avec `ref`,
- `name` : string ; incompatible avec `ref`,
- `ref` : Qname (référence à un autre type d'attribut) ; facultatif (incompatible avec `form`, `name` et `type`),
- `type` : Qname (type simple de la valeur de l'attribut) ; incompatible avec `ref`,
- `use` : « `optional` », « `prohibited` » ou « `required` » ; facultative (valeur par défaut : « `optional` ») ; doit avoir pour valeur « `optional` » si une valeur par défaut est indiquée (i.e. si `default` est présent).

Remarque : la spécification d'attributs peut aussi être réalisée par l'intermédiaire de groupes d'attributs nommés, introduits par l'élément `<atrributeGroup>` ou par l'autorisation d'attributs non validés (élément `<anyAttribute>`), que nous ne détaillons pas ici.

La spécification de contenus se fait par l'intermédiaire de définitions d'un élément `<simpleContent>` ou d'un élément `<complexContent>`. Le premier ne peut pas contenir d'attribut et ne peut contenir qu'un élément indiquant le mode de dérivation utilisé (`<extension>` ou `<restriction>`). Le second peut avoir un attribut (`mixed`, qui indique la présence à la fois de caractères et d'éléments dans le contenu) et d'un élément `<extension>` ou `<restriction>`.

Les éléments `<extension>` et `<restriction>` permettent de spécifier des structures de contenus, introduits par les éléments `<group>`, `<all>`, `<choice>` et `<sequence>`.

2.5.2.7 Préambule d'un schéma

Le vocabulaire utilisé dans un schéma XML est défini dans un espace de noms spécifique, identifié dans un *préambule* (i.e. l'élément racine du schéma). Un schéma conforme à la recommandation du W3C a pour racine un élément `schema`, rattaché à l'espace de noms de spécification des schémas du w3C (pour lequel il est recommandé d'utiliser le préfixe « xsd », pour XML schema definition). La syntaxe valide est :

```
<xsd:schema
      xmlns:xsd="http://www/w3.org/2001/XMLSchema"
      xsd:targetNamespace="http://www.monsite.com/monnamespace">
```

L'attribut `targetNamespace` permet d'identifier l'espace de noms que le schéma définit. Tout document XML faisant référence à cet espace de noms et se déclarant conforme à la recommandation sur les instances de schémas XML doit respecter la structure décrite dans ce schéma pour être validé.

Un schéma peut cependant ne pas se référer à un espace de noms. Dans ce cas, on peut utiliser l'attribut `noTargetNamespace` (avec n'importe quelle valeur), ou omettre cet attribut. Sans espace de noms, la déclaration précédente devient alors :

```
<xsd:schema
      xmlns:xsd="http://www/w3.org/2001/XMLSchema"
      xsd:noTargetNamespace="noTargetNamespace">
```

ou simplement :

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

Remarques :

- Un schéma n'est pas nécessairement situé dans un fichier à part. Ce peut être un sous-élément de l'arborescence d'un document XML.
- Un schéma peut faire référence à des éléments d'autres schémas, par des mécanismes qui ne sont pas détaillés ici.

2.5.2.8 Association d'un document XML à un schéma

Pour qu'un document XML puisse être validé par un schéma, il faut déclarer ce document comme une *instance de schéma XML* (le préfixe recommandé est « xsi »). L'attribut `schemaLocation` permet la validation du document en fonction d'un schéma relatif à un espace de noms. La valeur de cet attribut est en deux parties : l'URI de référence du schéma (i.e. le « `targetNamespace` » du schéma) et le nom du fichier (avec éventuellement un chemin) contenant le schéma, séparés par une espace. Il faut aussi déclarer l'espace de noms auquel appartiennent les éléments du document XML à valider. La syntaxe correcte pour l'élément du document à valider est :

```
<ici:element
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://monsite.com/monnamespace
            http://monsite.com/monnamespace/schema/MonSchema.xsd"
      xmlns:ici="http://monsite.com/monnamespace">
```

Cette déclaration requiert la validation de `ici:element` et de tous ses sous-éléments et attributs préfixés par « `ici:` ».

Si le schéma ne fait référence à aucun espace de noms, on utilise l'attribut `noNamespaceSchemaLocation` avec pour valeur le fichier contenant le schéma :

```
<element
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://monsite.com/monnamespace/
schema/MonSchema.xsd">
```

2.6 Les espaces de noms XML

2.6.1 Position du problème

XML permet une grande liberté dans le choix des noms de balises utilisées pour structurer l'information. Des balises identiques peuvent être utilisées pour décrire différents types de méta-information, ce qui est parfaitement valide en XML. Par conséquent, des conflits peuvent apparaître lors de l'analyse et de l'interprétation (mise en forme) du document. Par exemple, les balises `<title>`, `<commande>`, `<liste>` peuvent être interprétées de façons très diverses (balises HTML ou titre de civilité, instruction ou sélection d'articles, liste d'éléments de natures très diverses…).

2.6.2 Principe général

Cette redondance est imputable à une *polysémie* des termes, et peut être levée par la donnée du contexte d'utilisation de ces termes. Les *espaces de noms XML* ont été créés pour différencier les mêmes termes utilisés dans le contexte de *vocabulaires* différents. Ces vocabulaires sont des regroupements de termes établis de manière purement conceptuelle. L'idée est bien entendu de définir ces regroupements de manière suffisamment fine pour ne pas avoir de conflits de termes à l'intérieur d'un espace de noms.

Remarque : les espaces de noms XML sont des vocabulaires et non pas des types de documents. Ils ne fournissent aucune information sur la signification ou la structure des méta-données.

2.6.3 Unicité des espaces de noms

Chacun de ces vocabulaires est alors défini de façon unique en le liant avec une URI (i.e. une URL ou une URN[22]). L'idée est de particulariser les balises en leur ajoutant l'URI comme préfixe.

Par exemple, la balise <title> correspondant à l'annonce d'un titre en XHTML pourrait être définie par `<{http://www.w3.org/1999/xhtml}title>`, ce qui permettrait par exemple de la différencier de celle de votre CV, définie par `<{http://www.iup.univ-avignon.fr/etds/CV/english}title>`. Bien entendu, ces balises ne sont pas correctes, puisqu'elles contiennent les caractères '{', '}' et '/', qui sont interdits dans les balises XML.

2.6.4 Syntaxe d'utilisation des espaces de noms XML

La syntaxe retenue par le W3C (disponible à l'adresse : http://www.w3.org/TR/REC-xml-names) est l'utilisation de *noms qualifiés* (Qnames). Ces noms qualifiés sont composés de deux parties : la *partie locale*, (les noms « simples » des éléments dans le fichier XML d'origine) et le *préfixe d'espace de noms*, indiquant l'espace de noms auquel l'élément appartient. Ce préfixe fait référence à un espace de noms désigné par une URI. Le préfixe est séparé de la partie locale du nom par le caractère ':'.

---

[22] Universal Resource Name, voir plus loin.

L'association du préfixe et de l'espace de noms est réalisée en utilisant un attribut commençant par le terme « xmlns: ». Cet identifiant doit respecter la syntaxe XML. L'attribut ainsi constitué reçoit pour valeur l'URI choisie (ici, des URL complètes, c'est-à-dire avec le nom du protocole utilisé et éventuellement les sous-répertoires du nom de domaine).

```
<?xml version="1.0"?>
<etd:CV xmlns:etd="http://www.iup.univ-avignon.fr/etds/CV/english"
        xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <etd:personne>
    <etd:civil_status>
      <etd:title>Mr.</etd:title>
    </etd:civil-status>
    ...
  </etd:personne>
  <xhtml:html>
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>
</etd:CV>
```

Remarques :

- On ne peut utiliser plusieurs URI différentes pour spécifier le même espace de noms (comme on le fait par exemple pour localiser une DTD), car cela conduirait à définir plusieurs valeurs pour le même attribut d'une balise XML.
- Les attributs XML peuvent également être associés à des espaces de noms, selon la syntaxe : `<nom_ns1:balise nom_ns2:attribut="valeur">`. Toutefois, toutes les applications ne reconnaissent pas le nommage des attributs XML. On tentera donc de ne pas l'utiliser.
- Le préfixe en lui-même n'a pas de signification. Son remplacement par n'importe quel autre préfixe n'altèrerait en rien la signification du document.
- Au niveau interne, lors de l'analyse d'un document, le processeur remplace simplement tous les préfixes d'espaces de noms par l'espace de noms lui-même (i.e. l'URI référencée). Cette syntaxe non valide utilisée en interne par le processeur est appelée *noms pleinement qualifiés*.

2.6.5 Espaces de noms par défaut

Un espace de noms par défaut fonctionne comme un espace de noms normal, hormis qu'il ne lui est pas associé de préfixe. Tous les éléments du document XML ne possédant pas de préfixe lui sont alors associés.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english"
    xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>
```

```
  <xhtml:html>
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>
</CV>
```

2.6.6 Espaces de noms descendants

L'attribut « xmlns: » de déclaration d'espaces de noms n'est pas nécessairement situé dans l'élément racine du document XML. Cet attribut peut être associé à tout autre élément regroupant l'ensemble des termes utilisés dans le vocabulaire. Dans certains cas, cela améliore même la lisibilité du document.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>
  <xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:head>
      <xhtml:title>CV of an IUP student</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    ...
    </xhtml:body>
  </xhtml:html>
</CV>
```

Il est même possible de définir des espaces de noms par défaut différents pour les éléments enfants et pour l'élément racine.

```
<?xml version="1.0"?>
<CV xmlns="http://www.iup.univ-avignon.fr/etds/CV/english">
  <personne>
    <civil_status>
      <title>Mr.</title>
    </civil-status>
    ...
  </personne>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>CV of an IUP student</title>
    </head>
    <body>
    ...
    </body>
  </html>
</CV>
```

2.6.7 Annulation des espaces de noms

Il est possible d'annuler l'appartenance d'un élément interne à un espace de noms en lui associant un espace de noms par défaut vide. On emploie alors l'attribut « `xmlns=""` ».

2.6.8 URI, URL, URN et signification des espaces de noms

Nous avons vu qu'un espace de noms est désigné par une URI (c'est-à-dire une URL ou une URN complètes). L'objectif est de définir une ressource conceptuelle permettant d'assurer l'unicité de l'espace de noms. Plusieurs raisons peuvent être avancées pour justifier le choix de l'une ou l'autre de ces notations.

D'un point de vue théorique, le choix d'une URN est plus « libre », et correspond mieux à la définition d'un espace de noms. En effet, la structure une URN se présente sous la forme suivante : « urn:NID:NSS », où NID (namespace identifier) représente le nom de l'espace de noms et NSS (namespace specific string, à vérifier) une chaîne spécifique à l'espace de noms. Les URN permettent donc de *nommer* une ressource et non pas de *pointer vers* une ressource, ce qui est plus conforme à l'esprit de la définition d'espaces de noms, tel qu'il est défini dans la recommandation du W3C.

D'un point de vue pratique, rien ne garantit l'unicité de la NSS d'une URN. L'utilisation d'URLs est donc un moyen simple d'éviter les conflits entre les espaces de noms.

Cependant, on pourrait être tenté d'aller plus loin dans la définition d'espaces de noms. En effet, la plupart des utilisateurs voudraient donner du sens aux noms des éléments. Cela permettrait notamment, dans l'optique de la structuration documentaire, de définir la signification de la méta-information présente dans les documents XML : à quoi cette méta-information sert-elle ? Quelles sont les différentes valeurs possibles pour les balises. S'il était possible de définir une description liée à l'espace de noms, cela serait sans doute un bon moyen de valider la conformité du document en fonction des espaces de noms utilisés. Cela pourrait même permettre d'éviter l'écriture de DTDs, parfois complexes, et souvent fastidieuses.

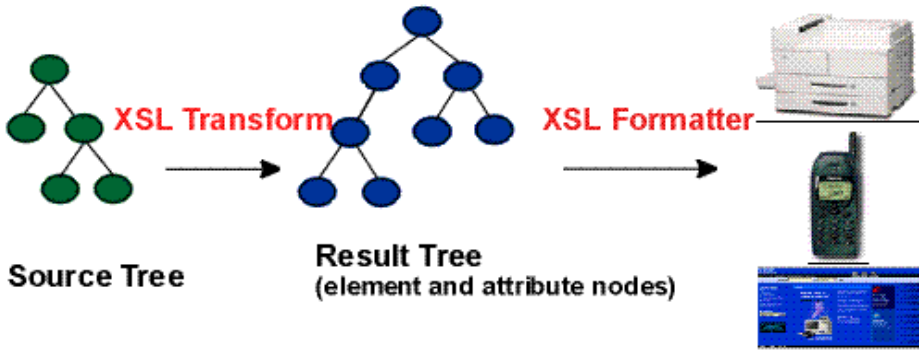
A l'heure actuelle, la spécification des espaces de noms XML du W3C indique que le rôle premier d'une URI d'espace de noms n'est pas de permettre la récupération d'une telle description. L'URI est uniquement un identificateur, qui n'a pas de sens en soi. En particulier, il ne signifie rien pour le processeur XML, qui le transmet tel quel à l'application.

Cependant, rien n'empêche le « propriétaire » de l'espace de noms (c'est-à-dire le premier à s'être attribué l'URI référencée) de fournir une description de cet espace qui sera utilisée pour la validation du document XML. Une telle description ne peut être fournie sous forme de DTD, car un document XML ne peut comporter qu'une seule DTD. L'utilisation de *schémas XML* est plus indiquée pour une telle description.

2.7 XSL (Extended Stylesheet Language)

2.7.1 Le processus de composition documentaire

XML et les autres langages abordés dans les paragraphes précédents abordent les différents aspects de la structuration documentaire. Lorsque l'on a créé un document XML bien formé et valide, on est en présence d'une structure de données incluant informations et méta-informations. Il s'agit alors de modifier et de mettre en forme cette structure de données dans un format documentaire adapté à l'utilisation envisagée. Le processus de production documentaire est alors illustré par la figure ci-dessous, extraite de la recommandation XSL du W3C du 15 octobre 2001 (http://www.w3.org/TR/xsl/).



Dans ce cours, il s'agit de générer des documents textuels lisibles par des opérateurs humains. Les formats visés peuvent être du texte simple (TXT) ou mis en forme (RTF), des formats documentaires structurés (PS, PDF) ou des formats adaptés à un média particulier, comme les pages web (HTML) ou les documents sonores (pages aurales). Rien n'empêche cependant de réaliser une telle mise en forme pour d'autres applications qui prendront en entrée des fichiers d'échange de données en XML dont la structure sera différente de celle des fichiers d'origine.

Le but de ce cours n'est pas de traiter tous ces formats. Il se limite aux cas simples de génération de documents textuels ou XHTML. Les techniques « standard » sont présentées dans le cadre d'un exemple d'application : la composition de pages web virtuelles. Il existe bien entendu d'autres techniques qui ne sont pas abordées ici. Notamment, la génération de documents complexes (comme le format pdf d'Adobe™), via le langage de description de pages XSL *Formatting Objects* (XSL-FO) associé à XSLT, dépasse le cadre de ce cours. Nous nous contentons de réaliser l'opération de formatage par l'application de feuilles de style CSS.

2.7.2 Exemple de technique de composition documentaire

L'objectif de cette partie est de présenter un ensemble de techniques de composition documentaire, connu sous le nom de *Documents Virtuels Personnalisables* (DVP). Ces techniques permettent de générer à la volée des fichiers XHTML visualisables dans un navigateur W3 (ici : Internet Explorer® 5). Ces fichiers correspondent à la notion de document, car ils possèdent une structure de données et des informations de formatage spécifiques. Ils sont virtuels, car générés à la volée et non stockés sur un support physique. Ils sont personnalisables, car la structure de données ainsi que les informations de formatage

peuvent être générées en fonction d'actions des utilisateurs, ce qui permet de créer des documents prenant mieux en compte leurs besoins que les documents « statiques ».

L'ensemble des opérations de mise en forme nécessite trois types d'opérations. Chacune d'elles est réalisée par un outil spécifique. Les outils indiqués ici sont ceux utilisés pour la composition de DVP.

Récupération des informations et méta-informations : cette opération permet d'inclure des éléments de la structure de données source dans la mise en forme du document final. Dans le cas d'une structure de données en XML, le langage utilisé pour parcourir cette structure s'appelle *XPath* (XML Path Language).

Transformation de la structure de données : il s'agit de convertir les informations et méta-informations sources (i.e. les parties de contenu du document XML d'origine, obtenues par l'opération précédente) en une structure de données de sortie correspondant au format choisi. Techniquement, cette étape s'apparente plus à la structuration qu'à la composition documentaire. Cependant, elle est intégrée au processus de composition car elle n'est pas mise en œuvre si ce processus n'est pas initié. Le langage utilisé pour transformer ces structures de données s'appelle *XSLT* (Extended Stylesheet Language Transformation).

Application des styles : c'est l'opération de composition documentaire proprement dite. La technique utilisée est très variable selon l'application (par exemple, dans les milieux de l'édition le langage utilisé est DSSSL, car il permet de spécifier avec une grande précision les caractéristiques des pages). Pour la composition de documents au format W3 (et *a fortiori* de documents XHTML), le standard recommandé par le W3C est *CSS*.

Remarques :

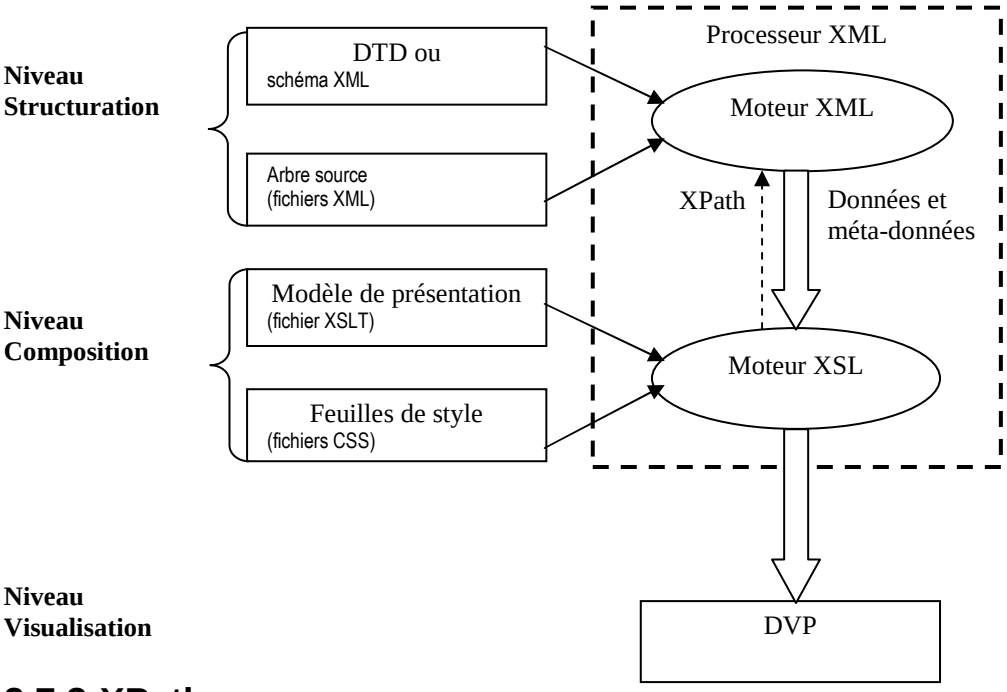
- L'association de XPath et de XSLT[23] constitue le langage de feuilles de styles « officiellement » rattaché à XML : *XSL* (Extended Stylesheet Language).
- La séparation des opérations de génération de documents XHTML visualisables avec XSLT et d'application de feuilles de style avec CSS permet de se conformer à la recommandation XHTML 1.0 stricte. Si des informations de style sont incluses dans le document XHTML généré en XSLT (i.e. dans la feuille de style XSLT), on obtient du XHTML transitionnel, et donc imparfait…
- Les navigateurs récents (IE5…) prennent en charge l'affichage direct de documents XML reliés à des feuilles de style CSS2, par l'instruction de traitement : `<?XML:stylesheet type="text/css" href="Feuille_De_Style.css"?>`
- La prise en charge des feuilles de style diffère selon l'implémentation qui en est faite dans le navigateur, c'est-à-dire en fonction du navigateur et de sa version.
- Chacune des opérations ci-dessus n'est théoriquement pas obligatoire. Cependant, le processus de composition documentaire privé d'une de ces étapes aboutit à des résultats plus ou moins utiles. Par exemple, l'utilisation des données de l'arbre XML source n'est théoriquement pas nécessaire : il est possible d'écrire une feuille XSLT ne faisant aucun cas du contenu du document XML d'origine. On peut cependant s'interroger sur l'intérêt d'utiliser des techniques aussi complexes ici pour générer des documents XHTML figés. Il est également possible de ne pas faire subir de transformations au document XML source. Par exemple, un fichier XHTML est un document XML directement visualisable

[23] Plus XSL-Formatting Objects.

avec un navigateur W3. Enfin, la génération de fichiers textes ou XML ne requiert pas l'application de feuilles de style.

Les langages de composition sont pris en charge par des applications spécifiques : les *moteurs XSLT*. Ces moteurs sont inclus dans certains processeurs XML. Ci-dessous, on distingue – artificiellement – le moteur XML du moteur XSL inclus dans un processeur XML. Par chance (!), MSXML contient un tel moteur. Un document XML associé à une feuille de style XSL peut donc être directement visualisé dans Internet Explorer™ 5.

Le processus global de génération de DVP est illustré dans la figure ci-dessous.



2.7.3 XPath

XPath est un outil souple permettant de pointer vers les différents morceaux d'un document XML. La syntaxe de ce langage permet de définir des *expressions XPath* qui sont utilisées :

- pour le parcours de documents XML,
- pour le test de valeurs associées aux contenus ou aux attributs d'éléments.

La syntaxe de ce langage est volontairement différente de celle de XML. Cela est dû au fait que les expressions XPath sont principalement utilisées dans des attributs d'éléments XSLT (qui eux respectent la syntaxe XML). La syntaxe XPath a donc été définie pour ne pas « perturber » l'analyse des feuilles de style XSLT par le parser XML. En particulier, elle ne fait – pratiquement – pas appel au caractère de début de balise '<'.

La recommandation du W3C date du 16 novembre 1999. Elle est disponible à l'URL : http://w3.org/TR/REC-xpath/.

2.7.3.1 Syntaxe élémentaire des expressions de parcours

Dans un premier temps, XPath permet de spécifier des *chemins de localisation* de parties de document. Pour cela, ce langage utilise le concept de *nœud*.

Un nœud XPath désigne une partie quelconque de l'arborescence d'un document, qu'il s'agisse d'un élément, d'un contenu ou d'un attribut. Une expression permet également de désigner des *ensembles de nœuds*, par des propriétés qu'ils ont en commun (par exemple, l'ensemble de tous les éléments possédant un attribut « id »). Des nœuds particuliers sont la *racine du document* et le *nœud contextuel*.

La racine du document désigne en XPath une entité qui ne pointe vers aucun des éléments du fichier XML et qui a pour enfant l'élément racine du document. Cette entité est désignée par le caractère '`/`' (slash) en début d'expression.

Le nœud contextuel est la position courante dans le document. La sélection d'un nœud contextuel est présentée plus loin. Ce nœud contextuel est désigné par '`.`' (point).

Un chemin de localisation XPath peut être absolu, relatif ou récursif. Dans le premier cas, il part de la racine du document. Dans le deuxième, il part du nœud contextuel. Dans le troisième cas, le moteur XSLT parcourt la totalité du document à la recherche de l'expression cherchée. L'opérateur de descente dans l'arborescence de l'arbre est désignée par le caractère '`/`' (slash). L'opérateur de descente récursive dans l'arborescence est désignée par les caractères "`//`" (double slash). Par défaut, un chemin de localisation est considéré comme relatif.

Remarque : l'opération de descente récursive a tendance à dégrader sérieusement les performances des feuilles de style. Quand cela est possible, il est préférable d'éviter de l'utiliser.

Un attribut est désigné en XPath par le caractère '`@`'. En XPath, un attribut est considéré comme un sous-nœud de l'élément XML auquel il se rapporte. Bien entendu, un chemin de localisation XPath comportant un opérateur de descente (récursive ou non) suivant ce caractère est invalide.

Il existe aussi un **caractère générique** "`*`" qui désigne n'importe quelle valeur ou élément non vides.

Exemples :

`/` désigne l'ensemble du document XML.

`./Etat_civil/Nom` (ou `Etat_civil/Nom`) désigne les éléments `<Nom>` enfants des éléments `<Etat_civil>`, eux-mêmes enfants du nœud contextuel.

`Etat_civil/*` désigne tous les éléments enfants des éléments `<Etat_civil>`, eux-mêmes enfants du nœud contextuel.

`/racine/@xmlns` désigne l'attribut `xmlns` (rappel : espace de noms par défaut) de l'élément racine du document.

`//xhtml:div/@id` désigne les attributs `id` de tous les éléments `<html:div>` où qu'ils soient situés dans l'arborescence du document XML.

`.//enfant` désigne tous les éléments `<enfant>` descendant du nœud contextuel.

2.7.3.2 Filtrage des chemins de localisation

Les exemples précédents permettent de sélectionner des nœuds en fonction de leur ascendance. Il est possible de définir des règles de sélection prenant également en compte la descendance des nœuds. Pour cela, la syntaxe XPath permet d'utiliser des *filtres de localisation*, encadrés par les caractères '`[`' et '`]`' (crochets). Ces filtres peuvent porter sur l'existence d'un nœud « enfant » dans la descendance du nœud à sélectionner ou sur la valeur d'un tel enfant.

Exemples :

`Etat_civil[Nom]` désigne tous les éléments `<Etat_civil>` enfants du nœud contextuel et ayant au moins un élément enfant `<Nom>`.

`Etat_civil[Nom = 'toto']` désigne tous les éléments `<Etat_civil>` enfants du nœud contextuel et ayant au moins un élément enfant `<Nom>` dont la valeur est « toto ».

`Etat_civil/Nom[. = 'toto']` désigne tous les éléments `<Nom>` dont la valeur est « `toto` », enfants d'éléments `<Etat_civil>`, eux-même enfants du nœud contextuel.

`//xhtml:div[@id]` désigne tous les éléments `<html:div>` où qu'ils soient situés dans l'arborescence du document XML, pourvu qu'ils aient un attribut « id ».

`//xhtml:div[@id = 'div1']` désigne l'élément `<html:div>` où qu'il soit situé dans l'arborescence du document XML, dont la valeur de l'attribut `id` est « `div1` » (rappel : un attribut `id` a par définition une valeur unique dans le document).

2.7.3.3 Les fonctions XPath

XPath possède un certain nombre de *fonctions* pouvant être utilisées dans des expressions XPath. Comme une expression XPath doit toujours renvoyer un nœud, les fonctions qui n'ont pas ce type de valeur de retour sont toujours utilisées à l'intérieur de filtres de localisation. Ces fonctions sont de la forme :

```
Nom_De_Fonction ( [Arguments] )
```

Il existe plusieurs types de fonctions.

Les fonctions de nœuds, qui permettent de travailler avec des nœuds en général. Ces fonctions sont : `name()`, `node()`, `processing-instruction()`, `comment()` et `text()`.

Les fonctions de position, qui permettent de déterminer ou d'identifier la position d'un nœud dans un ensemble de nœuds. Ces fonctions sont : `position()`, `last()` et `count()`.

Les fonctions numériques, qui ne sont pas énumérées ici. La plus couramment utilisée est `number()`, qui permet de convertir la valeur d'un nœud (PCDATA) en nombre. Par exemple : `article[number(prix) < 1000]` sélectionne tous les éléments `<article>` dont un élément enfant `<prix>` a pour valeur un nombre inférieur à 1000.

Les fonctions booléennes, dont les plus courantes sont `boolean()`, `not()`, `true()` et `false()`.

Les fonctions de traitement de chaînes, parmi lesquelles `string()`, `string-length()`, `concat()`, `contains()`, `substring()` ou encore `starts-with()`.

2.7.3.4 Axes de navigation

XPath ne permet pas seulement de se déplacer dans l'arborescence du document XML par des opérations de descente et de remontée. L'utilisation d'*axes XPath* permet des déplacements plus complexes dans cette arborescence. Les axes XPath sont la généralisation de la notion de chemin de localisation. Certains de ces axes possèdent des abréviations, qui sont celles que nous avons déjà vues dans la description des chemins de localisation. La syntaxe appropriée pour leur utilisation dans des expressions XPath est : `Nom_D'Axe::Nom_De_Nœud`.

Il existe 13 axes XPath.

Nom d'axe	Description	Exemple d'utilisation / syntaxe abrégée
`self`	Nœud contextuel	`self ::node()` ou `./node()`ou `.`
`child`	Enfants du noeud contextuel (par défaut)	`child::Etat_civil` ou `Etat_civil`
`descendant`	Tout enfant, petit-enfant etc. du nœud contextuel	`descendant::Etat_civil`
`descendant-or-self`	Comme descendant + le nœud contextuel lui-même	`descendant-or-self::Etat_civil` ou `.//Etat_civil`
`parent`	Parent du noeud contextuel	`//Nom/parent::Prenom` ou `//Nom/../Prenom`
`ancestor`	Tout parent, grand parent etc. du nœud contextuel	`ancestor::Prenom`
`ancestor-or-self`	Comme parent + le nœud contextuel lui-même	`ancestor-or-self::Prenom`
`following-sibling`	Tous les frères suivants du nœud contextuel (vide si le nœud est un attribut)	`following-sibling::Nom`
`preceding-sibling`	Tous les frères précédents du nœud contextuel (vide si le nœud est un attribut)	`preceding-sibling::Prenom`
`following`	following–sibling + descendants de tous les nœuds frères suivants.	`following::Nom`

`preceding`	preceding–sibling + descendants de tous les nœuds frères précédents.	`preceding::Prenom`
`attribute`	Attributs du nœud contextuel	`attribute::id ou ./@id`
`namespace`	Tous les nœuds appartenant au même espace de noms que le nœud contextuel	`namespace::xhtml:div`

2.7.4 XSLT

XSLT est un langage de programmation déclarative qui permet de décrire directement le contenu des fichiers de sortie, issus de la « transformation » des données et des méta-données XML. Sur le principe, une feuille de style XSLT est un document XML rattaché à un espace de noms spécifique qui définit des *balises de transformation* du contenu d'un document XML source à l'intérieur d'une série de *modèles* (templates) imbriqués. La recommandation du W3C concernant la version 1.0 de ce langage est datée du 16 novembre 1999 et est disponible à l'adresse : http://www.w3.org/TR/xslt.

L'élément racine d'une feuille de style XSLT est un élément `<stylesheet>`, qui a pour attributs la déclaration du numéro de version de XSLT utilisé et l'identification de l'espace de noms correspondant. Le préfixe recommandé pour cet espace de noms est « `xsl` ». Une déclaration « standard » de feuille de style XSLT est la suivante.

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

Remarques :

Les feuilles de style étant des documents XML, si elles sont dans des fichiers à part, elles n'échappent pas à la déclaration XML traditionnelle.

L'élément `<stylesheet>` est destiné à contenir la totalité des modèles XSLT de la feuille de style. Il n'est toutefois pas obligatoire, dans le cas où un seul modèle est défini dans cette feuille. Dans ce cas, il est possible d'indiquer la version de XSLT utilisée et l'espace de noms directement à la racine de l'arbre destination :

```
<racine_dest    xsl:version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Bien entendu, ce genre de simplification n'est pas recommandé pour la « compréhensibilité » du code.

En TP, le processeur XML utilisé (i.e. de MicroSoft™) peut aussi être activé par la déclaration suivante :

```
<xsl:stylesheet xmlns:xsl="uri:xsl">
```

Les éléments enfants directs de `<xsl:stylesheet>` sont appelés *éléments de niveau supérieur*. En théorie, il ne peut y avoir que deux types d'éléments de niveau supérieur : une balise `<xsl:output>` et des modèles. En pratique, un modèle peut être implicite, si bien que des éléments enfants du modèle peuvent être enfants d'une balise `<xsl:stylesheet>`.

<xsl:output> est une balise vide qui permet de spécifier le type de document généré en sortie de la feuille de style. Cela est réalisé par l'attribut method de cette balise, qui peut prendre « en standard » les valeurs xml, html ou text. D'autres valeurs spécifiques peuvent être acceptées par certains processeurs. En fonction de la valeur de l'attribut method, d'autres attributs peuvent être spécifiés. Ces attributs sont : version, encoding, omit-xml-declaration, standalone, cdata-section-elements ou indent. De tels attributs permettent par exemple de générer la déclaration XML dans le cas où ce format de sortie a été choisi par l'attribut method.

Remarque : le processeur MSXML étant destiné à visualiser les documents XML à l'intérieur d'Internet Explorer©, celui-ci n'admet qu'une seule valeur de l'attribut method : html.

2.7.4.1 Les modèles XSLT

Hormis la balise <xsl:output>, une feuille de style XSLT est constituée d'une succession de modèles XSLT. Chaque modèle spécifie ce qui doit être recherché dans l'arbre source et ce qui doit être placé dans l'arbre cible. Les définitions de modèles ne peuvent s'imbriquer les unes dans les autres. En revanche, il est possible d'appeler un modèle depuis la définition d'un autre.

Le contenu d'un modèle est encadré par une balise <xsl:template> qui possède un attribut match. La valeur de cet attribut est une expression XPath qui sert à sélectionner un ensemble de nœuds dans le modèle XPath de l'arbre source. Le modèle décrit peut alors s'appliquer à chacun de ces nœuds. Dans la description du modèle, on considère tout nœud correspondant à la valeur de l'attribut match comme le nœud contextuel à partir duquel est défini le modèle.

Le « modèle de départ » de toute feuille de style XSLT est associé à la racine XPath du document. En d'autres termes, la racine du document transformé doit être décrite à l'intérieur d'une balise <xsl:template match="/">. Lorsqu'aucun modèle n'est spécifié, le processeur utilise le *modèle caché par défaut* défini dans la balise ci-dessus.

Lorsque l'attribut match ne suffit pas à déterminer quel modèle doit être appliqué, il existe un attribut mode, permettant de différencier des modèles s'appliquant aux mêmes types de contenus. Dans ce cas, il faut préciser, lors de l'application du modèle, le mode d'application choisi. Il est également possible de laisser le processeur choisir un modèle à appliquer. Par défaut, lorsque deux modèles sont applicables, le processeur applique le dernier dans l'ordre de la feuille de style, sauf si :

l'un est plus spécifique que l'autre ; c'est notamment le cas quand un filtre XPath correspondant à un élément de l'arbre source a été utilisé dans l'un des attributs match des balises de définition des modèles,

une priorité est explicitement spécifiée par un attribut priority ; dans ce cas, c'est le modèle de plus forte priorité qui est choisi.

Remarque : l'avantage de la seconde solution est que la valeur de cet attribut est modifiable dynamiquement, comme celle de tout document XML, par exemple via le DOM.

L'application de modèles XSLT se fait avec la balise `<xsl:apply-templates>`. Cette balise permet alors d'appeler un ou plusieurs modèles depuis la description d'un autre modèle. Pour cela, la sélection du ou des modèles à appeler se fait grâce à une expression XPath dans un attribut `select`. Cette expression est évaluée, et le résultat est utilisé comme nœud contextuel s'il correspond à l'attribut `match` d'une balise `<xsl:template>`. L'attribut `mode` de la balise `<xsl:apply-templates>` peut être utilisé pour spécifier un modèle particulier, comme indiqué plus haut.

Remarques :

le contenu de l'attribut `select` est plus précis que celui de l'attribut match ; il permet de spécifier un chemin d'accès à un nœud, alors que celui de `match` détermine la condition d'application du modèle,

l'attribut `select` est facultatif ; dans le cas où il est omis, le processeur applique les modèles sur les balises, les PCDATA, les commentaires et les instructions de traitement contenus dans la balise courante,

dans tous les autres cas, l'appel de modèles peut être récursif ; les processeurs doivent être capables de gérer cette récursivité, et de détecter d'éventuels problèmes de boucles infinies.

Exemple :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/ Transform">
     <xsl:template match="/">
          ...
          <xsl:apply-templates select="/CV/Etat_civil/Nom"/>
     ...
     </xsl:template>

     <xsl:template match="Nom">
          Nom trouvé
     </xsl:template>
</xsl:stylesheet>
```

L'exemple ci-dessus affichera le texte « `Nom trouvé` » à chaque fois qu'il trouvera une balise `<Nom>`, enfant d'une balise `<Etat_civil>`, enfant de `<CV>`, lui-même enfant de la racine XPath du document source.

Les modèles nommés permettent l'appel d'un modèle indépendamment de son contenu. Pour cela, il suffit d'utiliser l'attribut `name` dans la balise de déclaration du modèle `<xsl:template>` et dans l'instruction d'appel `<xsl:call-template>`.

2.7.4.2 La génération de contenus XHTML

Pour produire une page XHTML virtuelle, la première chose à faire st de pouvoir générer des éléments XHTML. Dans la grande majorité des cas, cela se fait en écrivant tout simplement l'élément entre chevrons dans le modèle XSLT où il doit apparaître. On peut alors, si besoin est, utiliser les techniques de traitement des contenus XML présentées au paragraphe suivant.

L'élément <xsl:element> permet de réaliser exactement la même opération de façon plus compliquée. Par exemple, l'expression :

```
<xsl:element name="p">
  paragraphe
</xsl:element>
```

sera transformée en

```
<p> paragraphe </p>
```

par le processeur.

L'intérêt d'une telle démarche n'est pas flagrant *a priori*. Il le devient cependant, dans le cas où le nom de l'élément XHTML à générer dépend de la structure du document XML. En effet, l'attribut name de l'élément <xsl :element> peut contenir une expression de localisation XPath. Par exemple, si j'ai dans un fichier XML une énumération de balises HTML de la forme :

```
<balises_de_titre>
  <h1 />
  <h2 />
  <h3 />
  <h4 />
  <h5 />
  <h6 />
</balises_de_titre>
```

l'application des modèles suivants :

```
<xsl:template match="balises_de_titre">
  <xsl:apply-templates select="*">
</xsl:templates>

<xsl:template match="*">
  <xsl:element name="{.}">Niveau de titre trouvé</xsl:element>
</xsl:templates>
```

va mettre en forme la phrase « Niveau de titre trouvé » selon les 6 niveaux de titres HTML.

Remarque : l'intérêt de l'élément <xsl :element> est encore plus flagrant lorsque l'on n'utilise pas XSLT pour générer du HTML, mais pour transformer l'arbre source XML en un autre arbre XML. Dans ce cas-là, on se heurte sans arrêt au problème de la génération d'éléments dont on ne maîtrise pas le nom.

L'élément <xsl:attribute> permet exactement la même démarche pour la génération d'attributs. Il se place avant le contenu de l'élément auquel il se rapporte. Par exemple, l'expression :

```
<p>
  <xsl:attribute name="id">toto</xsl:attribute>
  paragraphe
</p>
```

sera transformée en :

```
<p> paragraphe </p>
```

par le processeur.

<xsl:attribute> présente cependant un intérêt supplémentaire par rapport à <xsl:element>, qui est qu'il permet de « sortir » la valeur de l'attribut d'entre les chevrons d'ouverture de la balise, ce qui permet d'utiliser les techniques de traitement des contenus XML (présentées ci-dessous) pour la génération de cette valeur. Par exemple, avec :

```
<lien>
  <destination>http://www.univ-lyon1.fr</destination>
  <texte>Lien vers le site de l'Université</texte>
</lien>
```
et la séquence XSLT :
```
<xsl:template match="lie">
  <a>
    <xsl:attribute name="href">
      <xsl:value-of select="destination" />
    </xsl:attribute>
    <xsl:value-of select="texte" />
  </a>
</xsl:template>
```
le processeur génère :
```
<a href="http://www.univ-lyon1.fr">
  Lien vers le site de l'IUP
</a>
```

2.7.4.3 Le traitement des contenus XML

La principale balise permettant d'accéder au contenu textuel d'un élément XML est la balise `<xsl:value-of>`. Elle s'emploie avec l'attribut `select`, qui définit le nœud XPath dont le contenu sera affiché. En remplaçant la ligne : « `Nom trouvé` » dans l'exemple ci-dessus par la ligne : « `Nom : <xsl:value-of select="."/>` », on obtient, pour chaque élément `<Nom>` correctement situé dans l'arbre source, le texte « `Nom :` » suivi du contenu de cet élément.

L'élément `<xsl:copy-of>` permet de recopier des parties entières de l'arbre source (y compris les attributs et les enfants). Il s'utilise avec l'attribut `select`, suivi comme il se doit d'une expression XPath. Il est principalement utilisé lorsque le format de sortie de la feuille de style est XML.

L'élément `<xsl:copy>` s'utilise à partir du nœud contextuel et permet un traitement plus riche qui n'est pas détaillé ici.

2.7.4.4 Les structures de contrôle XSLT

Les balises de contrôle XSLT permettent de réaliser plusieurs types de tests sur le contenu de l'arbre source. En fonction des résultats de ces tests, elles permettent de spécifier différents blocs de contenus pour l'arbre transformé.

L'élément `<xsl:if>`, employé avec l'attribut `test` permet de spécifier un contenu conditionnel, comme dans l'exemple suivant :
```
<xsl:if test="./Nom">
     Nom : <xsl:value-of select="Nom"/>
</xsl:if>
```

Cet exemple affichera « `Nom :` » et la valeur du nœud `<Nom>` uniquement si un tel nœud existe parmi les enfants du nœud contextuel.

L’élément `<xsl:choose>` permet de définir plusieurs alternatives. Il contient des éléments `<xsl:when>` définissant chacun une condition dans un attribut `test` et éventuellement un élément `<xsl:otherwise>` qui indique un contenu valide si aucun des tests des éléments `<xsl:when>` précédents n’a été évalué comme `true`.

L’élément `<xsl:for-each>` suivi de l’attribut `select` permet de définir une mise en forme à appliquer à tout élément correspondant à l’expression XPath. En d’autres termes, il est équivalent à l’élément `<xsl:apply-templates>`, mais rend la structure de la feuille de style moins lisible. On lui préfèrera donc ce dernier élément.

3ème PARTIE :

LA PROGRAMMATION AVEC XML

3.1 Introduction

3.1.1 Généralités sur les parsers

Nous avons vu qu'un parser (ou processeur) XML est un programme qui permet d'analyser et de modifier le contenu d'un document XML et éventuellement de valider ce contenu par rapport à une DTD ou un schéma XML. Un parser peut être considéré comme une boîte noire, qui s'interface entre un document et l'application qui a besoin d'y accéder. Il reçoit des requêtes de l'application, les exécute en analysant, parcourant ou modifiant le document XML, et renvoie un résultat ou une erreur à l'application[24]. Ce fonctionnement est tout à fait comparable à celui d'un SGBD, qui s'interface entre une application et une base de données.

On désigne par le terme API (Application Programming Interface) un protocole de communication entre deux programmes. Notamment, il existe des API permettant le dialogue entre les applications utilisant les documents XML et les parsers. Le schéma général de communication via ces API est illustré dans la figure ci-dessous.

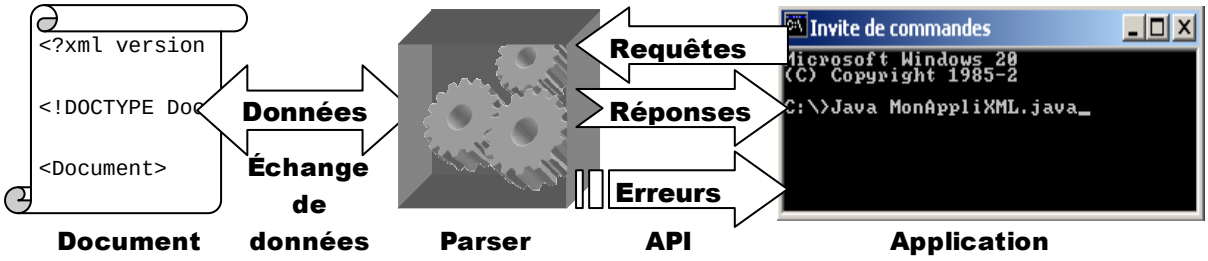


Figure 2. Schéma général d'utilisation d'un document XML dans une application

Bien entendu, il existe un grand nombre de parsers XML, et leurs API varient en fonction des langages de programmation dans lesquels ils sont écrits et de leurs implémentations. Par exemple, avec le parser AElfred de Microstar Software[25], pour programmer en Java avec XML, il faut importer le package contenant ce parser (`com.microstar.xml`), travailler dans une classe implémentant une classe spécifique à l'API du parser (`XmlHandler`), puis déclarer une instance de la classe représentant un parser XML (`XmlParser`), dont on peut appeler la méthode `parse()`. Un exemple de code utilisant AElfred contient donc les lignes suivantes :

```
import com.microstar.xml.*

...
```

[24] Les parsers HTML inclus dans les navigateurs fonctionnement sur un modèle similaire, à ceci près que la spécification de HTML indique qu'ils ne doivent pas renvoyer d'erreur lors de la lecture du document HTML, mais en ignorer les parties incorrectes.

[25] Ce parser a été implémenté par David Megginson, qui a également dirigé l'implémentation de SAX. C'est pourquoi l'API de AElfred ressemble à celle de SAX.

```
public class MonAppliXml implements XmlHandler {

...

XmlParser parser = new XmlParser();

...

parser.parse(...);
```

Au fur et à mesure de l'utilisation de XML dans les applications, deux standards ont fini par s'imposer : l'un, par la voie habituelle du W3C et l'autre par le biais d'une liste de diffusion pour développeurs XML. Ces standards sont le Document Object Model (DOM) et Simple API for XML (SAX), qui sont désormais implémentées par la plupart des parsers.

3.1.2 Java et XML : JAXP

Cela n'empêchait pas chaque application d'être conçue pour fonctionner avec un processeur particulier, malgré le fait qu'elle utilisait une API standard. En particulier, en Java, le grand nombre de parsers XML disponibles impliquait qu'à chaque installation d'une nouvelle application utilisant XML, il fallait installer le parser avec lequel cette application avait été développée. Il était alors fréquent de trouver plusieurs parsers installés sur la même machine virtuelle, pour faire exactement la même chose. Cela désobligeait l'ensemble de la communauté des développeurs Java, qui s'est sentie obligée de remédier à cette intolérable redondance. C'est pour cela qu'a été créée JAXP.

Initialement (i.e. en version 1.0), JAXP signifie « Java API for XML Parsing » (API Java pour l'analyse XML) et est disponible sous forme de package additionnel au JDK 1.3. JAXP 1.0 est structurée en une couche d'intégration des parsers et une couche API. La couche d'intégration permet d'utiliser n'importe quel parser installé sur une machine virtuelle pour traiter des documents XML, et n'importe quelle application qui utilise l'une des API de la couche API peut donc instancier et faire appel à ce parser de façon transparente. En version 1.0, la couche API supporte uniquement les deux types d'API pour XML 1.0 : le DOM (level 1) et SAX (1.0) et les espaces de noms XML.

Depuis, la signification du sigle a été changée pour « **Java API for XML Processing** » (API Java pour le traitement XML) et JAXP est incluse aux distributions Java standard. Ce changement de nom illustre la généralisation de JAXP, non seulement pour l'accès aux parsers, mais aussi pour la prise en charge de tous les traitements XML.

Début mars 2004, la dernière version de la spécification de JAXP est la 1.2 et est intégrée au JDK depuis les versions 1.4. La dernière « implémentation de référence » en date est la version 1.2.4[26] et est incluse dans le pack JWSDP (« Java Web Services Developer Pack ») 1.3.Elle inclut les parsers Xerces (pour l'analyse XML) et Xalan (comme processeur XSLT).

---

[26] Les spécifications sont des documents textuels qui expliquent la façon dont est censé fonctionner la version décrite. Les implémentations sont les versions des logiciels effectivement distribuées, qui [tentent de] se conformer à ces spécifications. Le numéro de version d'une implémentation commence par celui de la spécification à laquelle il se réfère, auquel il rajoute un chiffre correspondant à celui de la version de l'implémentation.

Elle prend désormais en charge le DOM level 2 et SAX 2.0, ainsi la transformation d'arbres XML via XSLT, grâce à TrAX (Transformation API for XML), et le support des schémas XML. La couche d'intégration des parsers a naturellement été aussi enrichie. Pour plus d'informations sur ces fonctionnalités, voir http://java.sun.com/xml/jaxp et http://java.sun.com/j2se/1.4/docs/guide/xml/jaxp.

Concrètement, JAXP fournit six packages contenant tout ce qui est nécessaire pour programmer en Java avec XML : `javax.xml.parsers`, `org.w3c.dom`, `org.xml.sax`, `org.xml.sax.helpers`, `org.xml.sax.ext` et `javax.xml.transform`. Dans tous les cas, il faut importer le premier package, contenant l'implémentation du parser qui sera utilisé, puis on travaille avec l'un ou plusieurs des quatre autres, selon que l'on veut utiliser le DOM, SAX et/ou XSLT. Le premier package est décrit ci-dessous. Les quatre suivants sont décrits dans les sections suivantes, et le dernier n'est pas abordé dans ce cours (il fonctionne cependant sur le même modèle).

3.1.3 Le package `javax.xml.parsers`.

Ce package définit deux types de classes abstraites symétriques pour chacune des API DOM et SAX : une « usine à parsers » (parser factory : `DocumentBuilderFactory` pour le DOM et `SAXParserFactory`, pour SAX 1.0), qui permet de créer des instances de parsers, issues d'une classe représentant un parser compatible avec l'implémentation choisie (`DocumentBuilder` et `SAXParser`). Dans les deux cas, il existe une méthode `parse()` qui lance l'analyse d'un document. Ce package définit aussi une erreur (`FactoryConfigurationError`) et une exception (`ParserConfigurationException`).

L'utilisation de ce package consiste à déclarer un objet « factory » correspondant à l'API choisie en utilisant la méthode `newInstance()` de l'une ou l'autre des deux classes, et de s'en servir pour créer une instance de parser accessible via cette API. Ensuite, il suffit d'appeler la méthode `parse()` de cet objet. Ci-dessous, un exemple de code DOM :

```
/* @(#)DomEcho01.java1.9 98/11/10
 *
 * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */
```

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import java.io.File;
import java.io.IOException;

import org.w3c.dom.Document;
import org.w3c.dom.DOMException;

public class DomEcho01{
    // Global value so it can be refered by the tree-adapter
    static Document document;

    public static void main(String argv[])
    {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        try {
           DocumentBuilder builder = factory.newDocumentBuilder();
           document = builder.parse( new File(argv[0]) );
        }
        catch (SAXException sxe) {
           // Error generated during parsing)
           Exception  x = sxe;
           if (sxe.getException() != null)
               x = sxe.getException();
           x.printStackTrace();
        }
        catch (ParserConfigurationException pce) {
            // Parser with specified options can't be built
            pce.printStackTrace();
        }
        catch (IOException ioe) {
           // I/O error
           ioe.printStackTrace();
        }
    } // main
```

Cet exemple est disponible en ligne à l'URL :

http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/dom/work/DomEcho01.java. Il indique les éléments de code nécessaires à l'utilisation du DOM. On remarque sur cet exemple que la méthode (surchargée) `parse()` de `DocumentBuilder` peut lever des exceptions de type `SaxException` ; c'est pourquoi il faut importer cette classe (et la classe `SaxParseException` qui en dérive).

En ce qui concerne SAX, l'implémentation minimale est un peu plus lourde et s'est compliquée au fur et à mesure des versions. L'exemple suivant n'en indique donc que la partie correspondant au lancement de l'analyse :

```
/* @(#)Echo01.java 1.5 99/02/09
 *
 * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY
 * IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR
 * NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE
 * LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS
 * LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF
 * OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or in
 * the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */

import java.io.*;

import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class Echo01 extends DefaultHandler
{
    public static void main(String argv[])
    {
        // Use an instance of ourselves as the SAX event handler
        DefaultHandler handler = new Echo01();
        // Use the default (non-validating) parser
        SAXParserFactory factory = SAXParserFactory.newInstance();
        try {
            // Parse the input
            SAXParser sp = factory.newSAXParser();
            sp.parse( new File(argv[0]), handler);
...
    }
```

L'intégralité de cet exemple est disponible en ligne à l'URL :

http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/work/Echo01.java. Attention, certaines des classes et des interfaces utilisées dedans ne sont plus valables (« deprecated ») pour SAX 2.0. Les sections suivantes comparent puis présentent plus précisément les API DOM et SAX (et notamment sax 2.0) et les packages spécifiques correspondants.

3.1.4 Java et XML : et JDOM ?

Contrairement à ce que son nom semble indiquer, JDOM n'est pas une implémentation en Java du DOM. C'est en fait une API qui fournit une autre représentation arborescente d'un document XML, manipulable à partir de packages spécifiques. JDOM se veut plus « naturelle » à utiliser que le DOM ou SAX. JDOM reste toutefois compatible avec ces deux standards, ce qui en fait une surcouche supplémentaire au dessus de celles-ci. JDOM n'est donc pas incompatible avec JAXP, et peut traiter des données obtenues à l'aide d'interfaces d'une distribution JAXP standard.

Nous ne détaillons pas l'API JDOM dans ce cours, car elle ne propose pas d'originalité dans son concept de traitement du XML. En revanche, comme il est vrai qu'elle peut simplifier un certain nombre de traitements, vous êtes encouragés à aller voir de quoi il retourne. Vous pourrez trouver les détails des spécifications de JDOM à l'URL : http://www.jdom.org.

3.1.5 Différences entre le DOM et SAX

Si l'on veut illustrer la différence entre le DOM et SAX, on peut comparer le processus d'analyse d'un document XML entre DOM et SAX à la différence existant entre une carte routière et un itinéraire. Dans un cas, le fait de disposer de la carte permet de se déplacer dans toutes les directions pour rejoindre n'importe quel point. Cependant, la quantité d'informations présentes sur la carte est parfois très importante par rapport à l'utilisation qu'on souhaite en faire. À l'inverse, le fait de disposer uniquement d'un itinéraire routier (i.e. prendre telle route ; sortie n°X ; 3ème feu à droite…) est plus simple à gérer, mais ne permet pas de se positionner dans un référentiel fixe (i.e. de « savoir où on est »).

De la même manière, le DOM peut être vu comme une « carte » représentant un document XML, et SAX comme un itinéraire, qui indique quel traitement effectuer de façon ponctuelle à chaque « intersection » : le principe de l'analyse d'un document XML avec SAX est de générer différents types d'événements à chaque fois qu'un type particulier de structure est rencontré dans le document XML (i.e. début ou fin d'élément XML, attribut…).

Chaque solution possède ses avantages et ses inconvénients : comme une carte, le DOM permet de déterminer plusieurs parcours différents sur le même arbre XML. Il est même possible de naviguer dynamiquement dans l'arbre et de modifier cet arbre au gré de l'utilisateur de l'application faisant appel à cette API. En revanche, il nécessite le chargement en mémoire de l'ensemble de la carte du document, qui peut être extrêmement consommatrice de ressources si le document est volumineux.

De son côté, SAX permet de déterminer facilement des itinéraires répétitifs, mais devient vite complexe à manipuler dans le cas de traitements multiples ou spécifiques. Il utilise cependant beaucoup moins de ressources que le DOM, car l'ensemble du document n'a pas besoin d'être chargé en mémoire pour réaliser ces traitements.

En résumé, si vous souhaitez effectuer différents types de parcours dans de petits documents XML, utilisez le DOM ; si vous avez des traitements répétitifs à faire sur des gros documents, utilisez SAX ; dans les cas intermédiaires, c'est à vous de voir. Par ailleurs, il existe également des approches mixtes, utilisant SAX pour localiser des parties d'arbres XML et le DOM pour traiter ces parties dans leur intégralité. Une telle approche n'est pas détaillée dans

ce cours, mais n'est pas très difficile à mettre en œuvre. De plus, de nombreux exemples de code sont disponibles sur le Web.

3.2 Le DOM (modèle objet de document)

3.2.1 Principe général

Le DOM a été créé puis standardise par le W3C pour permettre la manipulation de documents XML et HTML depuis des langages de programmation destinés au web tels que les langages de scripts ou Java. L'objectif était de rendre les applications w3 à la fois dynamiques (dans la lignée du DHTML) et portables.

Comme son nom l'indique, le DOM est un modèle objet, qui définit une API fondée sur la notion d'arbre[27] (« tree-based »). Il prend en charge la création, la modification et l'accès à des documents XML et HTML dont chaque partie (éléments et attributs) sont représentés sous forme d'objets correspondants aux nœuds de l'arbre. Le DOM fait correspondre, à chaque nœud d'un document XML, une *interface*[28] possédant des méthodes et des propriétés prédéfinies. Les objets d'un document XML (ou HTML) instancient ces interfaces.

Du fait que le DOM est une API (interface entre l'application et le parser qui analyse le document), le programmeur qui utilise le DOM ne travaille pas directement sur le document XML source, mais sur les objets représentant ce document. En théorie, il n'a jamais besoin de connaître le parser utilisé, car celui-ci se situe à un niveau inférieur et est masqué par le DOM.

Remarques :

- le DOM s'interfaçant entre l'application et le parser (et non le document directement), une condition essentielle à l'utilisation du DOM est que le document XML soit bien formé (et valide, si le parser utilisé est validant). De même, pour HTML, le DOM ne permet l'accès qu'à des documents valides. Un document HTML non valide est ignoré par le DOM.
- Il n'est pas tout-à-fait exact que le DOM permet d'ignorer le parser utilisé. La connaissance du fait que ce processeur est validant ou non, de son comportement par rapport aux caractères d'espacement multiples, ou des jeux de caractères qu'il prend en charge est nécessaire. Elle permet soit d'obtenir les données correspondant au document en sortie du parser, soit de prévoir la forme dans laquelle ces données seront renvoyées.

3.2.2 Spécifications

Le « DOM Working group » du W3C a débuté ses activités en août 1997. Une vue de l'état d'avancement des activités de ce groupe de travail est disponible à l'adresse : http://www.w3.org/DOM/Activity. Les figures présentées dans les paragraphes suivants sont extraites de cet état d'avancement.

Chacune des recommandations du DOM est décomposée en *modules*. Ces recommandations successives définissent des *niveaux* (DOM Level 1, 2 et 3) qui ont pour but de :

---

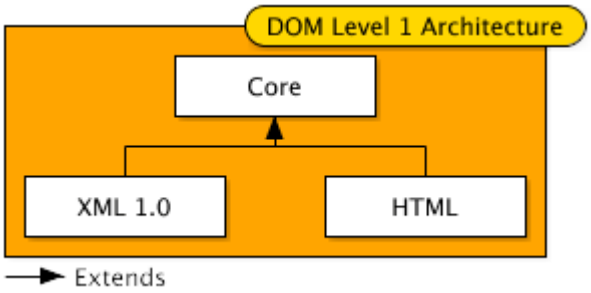
[27] Par opposition à SAX, qui s'appuie sur celle d'événement (« event-based »).

[28] Ces interfaces peuvent être vues comme des « classes abstraites » en C++.

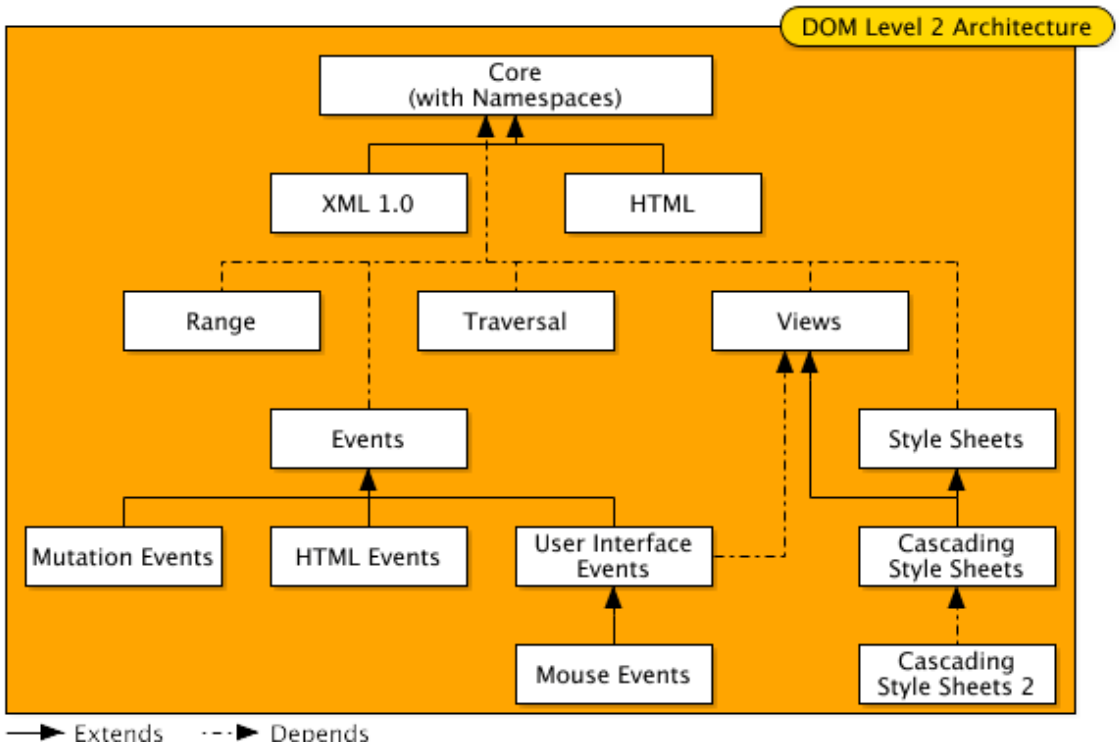
- corriger d'éventuelles erreurs ou imperfections de la version précédente,
- proposer de nouveaux modules pour la prise en charge de nouvelles propriétés.

Remarque : on peut également trouver des références au **DOM niveau 0**. Il s'agit des quelques fonctionnalités de manipulations des documents qui sont implémentées depuis la version 3.0 des navigateurs Netscape™ et Internet Explorer®. Aucune recommandation du W3C ne couvre ces fonctionnalités.

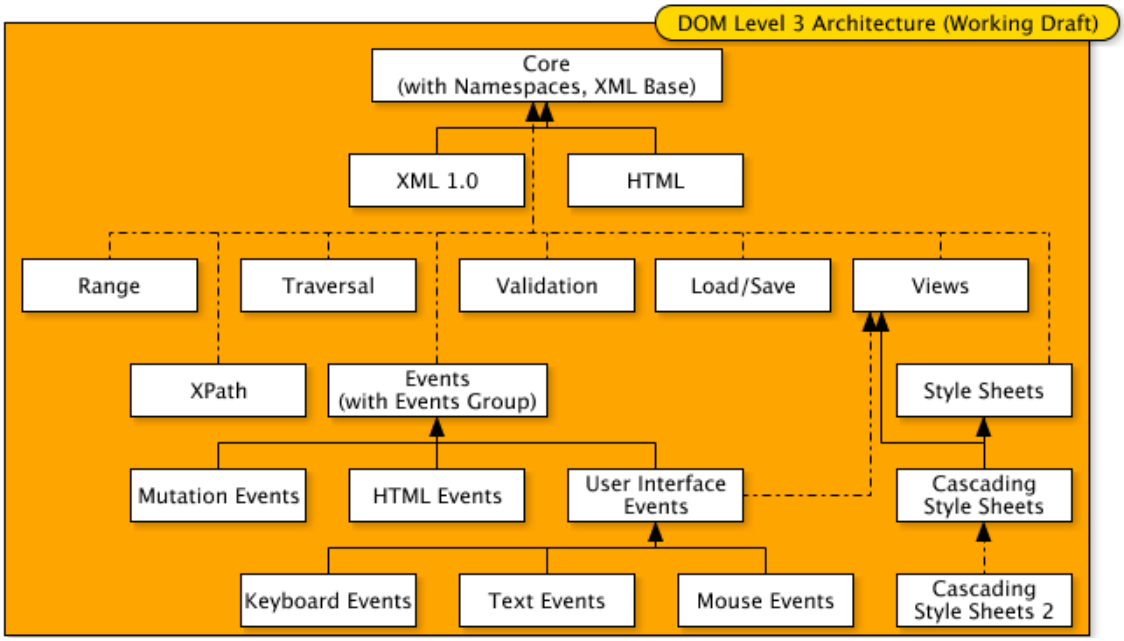
La recommandation du **DOM Niveau 1** date d'octobre 1998. Elle ne définit que trois modules. Un noyau et deux modules spécifiques à HTML et XML. Les fonctionnalités de ces modules sont décrites plus loin.



La première recommandation du **DOM niveau 2** date du 13 novembre 2000. Elle est disponible à l'adresse : http://www.w3.org/TR/DOM-Level-2-Core/ (pour le module Core). Elle enrichit le niveau 1 en ajoutant notamment des modules de prise en charge d'évènements et en améliorant les mécanismes de « range » et de déplacement dans les arbres. Le module Core accepte désormais les noms qualifiés (i.e. faisant référence à des espaces de noms). Il existe 5 nouvelles recommandation du DOM niveau 2 : *views*, *event*, *styles*, *traversal* et *range*, dont la dernière date de janvier 2003.



La plupart des recommandations officielles du **DOM niveau 3** sont au stade de « working group note » (quasiment finalisées). Seule la partie validation de documents est actuellement au stade de recommandation. Le niveau 3 comprendra une prise en charge plus complète des espaces de noms que le niveau précédent, et introduit la notion de schémas abstraits (i.e. DTD ou schémas XML) liés à un document XML. Un module « Load and Save » fait également son apparition, ainsi que la prise en charge de XPath.



Les différents modules possèdent des propriétés (*features*) spécifiques. Le tableau suivant indique les propriétés définies par les principaux modules du niveau 3. Ces propriétés ne sont pas des objets du DOM. Elles ne sont utiles que dans des cas particuliers, (par exemple, en tant qu'argument de méthodes comme `hasFeature()`[29]).

Nom du module	Nom de la propriété
Core	Core
XML	XML
Events	Events
User interface events	UIEvents
Mouse Events	MouseEvents
Text Events	TextEvents
Mutation Events	MutationEvents
HTML Events	HTMLEvents
Load and Save	LS
Abstract Schemas Editing	AS-Edit
XPath	Xpath

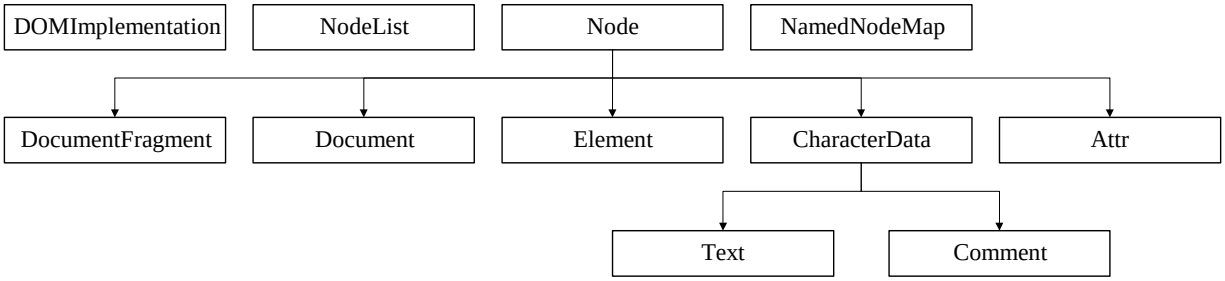
Remarque : les noms de propriétés sont insensibles à la casse (« case-insensitive »).

[29] Méthode qui renvoie `true` ou `false` si un module permet ou non l'accès à la propriété considérée.

3.2.3 Les différents modules du DOM

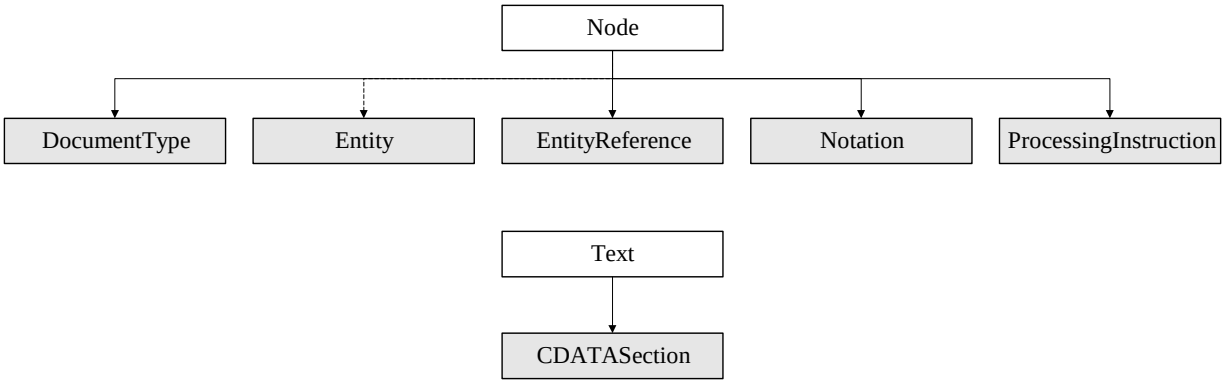
3.2.3.1 DOM Core

Le module Core définit une représentation interne arborescente d'un document. Il permet les déplacements dans ce modèle de document ainsi que l'accès et la manipulation d'objets du document. Il est également possible, grâce au module Core, de réaliser des manipulations sur la structure arborescente du modèle de document. Pour cela, le DOM Core propose des interfaces *fondamentales*. Ces interfaces sont définies soit dans le module Core lui-même, soit dans les modules dont il dépend (relations en pointillés sur les figures précédentes). Les interfaces fondamentales du DOM sont les suivantes (extraites de DOM Core niveau 2).



3.2.3.2 Le DOM XML

Il Le DOM propose aussi des interfaces *étendues* définies par les modules d'extension (relations en traits pleins) du module Core, c'est-à-dire les modules XML et HTML. Le DOM XML étend le module Core pour les besoins spécifiques au traitement de documents XML 1.0. Parmi ces spécificités, on compte la prise en compte des entités, des CDATA ou des instructions de traitement. Les interfaces étendues relatives au module XML sont en grisé sur la figure suivante.



3.2.3.3 DOM HTML

De la même façon, le DOM HTML étend le module Core pour les fonctionnalités spécifiques à HTML. Il n'est pas présenté ici. Ses spécifications sont disponibles à l'URL : http://www.w3.org/TR/DOM-Level-2-HTML.

Il existe aussi d'autres modules qui ne figurent pas sur les diagrammes précédents, concernant la prise en charge d'applications XML spécifiques comme MathML 2.0, SVG 1.0 ou SMIL Animation.

3.2.4 Interfaces du DOM

Chaque interface est un objet qui peut posséder des propriétés (*properties*[30]) et des méthodes. Celles-ci sont accessibles depuis un langage de programmation possédant une implémentation du DOM.

Remarque : un prototype peut également être associé à une interface ; dans ce cas, il définit des constantes spécifiques à cette interface, qui lui sont associées par le biais d'une propriété spécifique. Par exemple, la propriété `nodeType` de l'interface `Node` définit un ensemble de constantes représentant les différents types de nœuds existants.

3.2.4.1 Détail des interfaces

Les principales interfaces du DOM sont les interfaces `Node`, `Document` et `Element`. Les deux dernières sont des types particuliers de la première. Elles possèdent donc toutes les propriétés et méthodes de l'interface `Node` en plus de propriétés et de méthodes spécifiques.

L'interface `Node` permet d'instancier tous les nœuds de l'arborescence, qu'ils correspondent à des éléments XML, des attributs, des instructions de traitement ou d'autres types d'objets. Elle possède toutes les propriétés et méthodes élémentaires qui permettent de caractériser un nœud (comme un nom et une valeur), de se déplacer dans l'arborescence ou de manipuler celle-ci.

L'interface `Document` est à la racine du modèle de document. Elle permet non seulement d'accéder à l'arborescence des éléments XML, mais également à l'environnement défini à l'extérieur de l'élément racine XML. Par exemple, l'interface `Document` définit les attributs `version`, `encoding` et `standalone` de la déclaration XML, ainsi que différentes parties de la DTD (ou du schéma XML) éventuellement associé(e) au document[31].

L'interface `Element` désigne une balise du document XML ou HTML. En plus des propriétés de `Node`, elle possède la propriété `tagName`, qui permet d'accéder au nom de la balise. Les méthodes « utiles » spécifiques à l'interface Element sont relatives au traitement des attributs des balises (`getAttribute`, `setAttribute`, `removeAttribute` ou `hasAttribute`).

La liste ci-dessous indique les constantes, propriétés et méthodes des différentes interfaces utilisables depuis les langages de scripts. Elle est extraite des spécification du DOM niveau 3, et date du 13 septembre 2001. Elle est disponible à l'adresse : http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/ecma-script-binding.html.
Le détail de l'utilisation de chacune des interfaces est donné dans la spécification du module Core Level 3 (http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/).

---

[30] Par opposition aux « features » définies pour les modules, voir plus loin.

[31] À l'heure actuelle, la dernière version des spécifications du module Core est plus ancienne que celle du module Abstract Schemas. C'est pourquoi le W3C précise dans la première qu'il n'a pas indiqué les modalités de prise en compte des schémas abstraits depuis la propriété `doctype` de l'interface `Document`.

Remarque : n'ayant pas encore le statut de recommandation officielle, cette liste d'interfaces peut ne pas encore avoir été entièrement implémentée selon les spécifications du W3C dans les navigateurs ou les applications XML utilisées en TP.

Prototype Object `DOMException`

The `DOMException` class has the following constants:

`DOMException.INDEX_SIZE_ERR`

This constant is of type **Number** and its value is **1**.

`DOMException.DOMSTRING_SIZE_ERR`

This constant is of type **Number** and its value is **2**.

`DOMException.HIERARCHY_REQUEST_ERR`

This constant is of type **Number** and its value is **3**.

`DOMException.WRONG_DOCUMENT_ERR`

This constant is of type **Number** and its value is **4**.

`DOMException.INVALID_CHARACTER_ERR`

This constant is of type **Number** and its value is **5**.

`DOMException.NO_DATA_ALLOWED_ERR`

This constant is of type **Number** and its value is **6**.

`DOMException.NO_MODIFICATION_ALLOWED_ERR`

This constant is of type **Number** and its value is **7**.

`DOMException.NOT_FOUND_ERR`

This constant is of type **Number** and its value is **8**.

`DOMException.NOT_SUPPORTED_ERR`

This constant is of type **Number** and its value is **9**.

`DOMException.INUSE_ATTRIBUTE_ERR`

This constant is of type **Number** and its value is **10**.

`DOMException.INVALID_STATE_ERR`

This constant is of type **Number** and its value is **11**.

`DOMException.SYNTAX_ERR`

This constant is of type **Number** and its value is **12**.

`DOMException.INVALID_MODIFICATION_ERR`

This constant is of type **Number** and its value is **13**.

`DOMException.NAMESPACE_ERR`

This constant is of type **Number** and its value is **14**.

`DOMException.INVALID_ACCESS_ERR`

This constant is of type **Number** and its value is **15**.

Object `DOMException`

Remarque : cet objet est à la fois une exception et une interface. Il peut être généré (« raised ») par des propriétés ou méthodes d'autres interfaces et est accessible par le DOM.

The `DOMException` object has the following properties:

`code`

This property is of type **Number**.

Remarque : cette propriété renvoie l'une des constantes du prototype ci-dessus.

Object `DOMImplementationSource`

The `DOMImplementationSource` object has the following methods:

`getDOMImplementation(features)`

This method returns a `DOMImplementation` object.

The `features` parameter is of type **String**.

Remarque : cette méthode n'apparaît pas dans le document de travail dont est extraite cette liste et la définition de l'objet auquel est s'applique est incohérente avec le reste du document. Je l'ai modifiée pour la faire correspondre aux spécifications du module Core.

Object `DOMImplementation`

The `DOMImplementation` object has the following methods:

`hasFeature(feature, version)`

This method returns a **Boolean**.
The `feature` parameter is of type **String**.
The `version` parameter is of type **String**.

`createDocumentType(qualifiedName, publicId, systemId)`

This method returns a `DocumentType` object.
The `qualifiedName` parameter is of type **String**.
The `publicId` parameter is of type **String**.
The `systemId` parameter is of type **String**.
This method can raise a `DOMException` object.

`createDocument(namespaceURI, qualifiedName, doctype)`

This method returns a `Document` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
The `doctype` parameter is a `DocumentType` object.
This method can raise a `DOMException` object.

`getInterface(feature)`

This method returns a `DOMImplementation` object.
The `feature` parameter is of type **String**.

Object `DocumentFragment`

`DocumentFragment` has all the properties and methods of the `Node` object.

Object `Document`

`Document` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Document` object has the following properties:

`doctype`

This read-only property is a `DocumentType` object.

`implementation`

This read-only property is a `DOMImplementation` object.

`documentElement`

This read-only property is a `Element` object.

Remarque : cette propriété permet d'accéder à l'élément racine d'un document XML.

`actualEncoding`

This property is of type **String**.

`encoding`

This property is of type **String**.

`standalone`

This property is of type **Boolean**.

`strictErrorChecking`

This property is of type **Boolean**.

`version`

This property is of type **String**.

The `Document` object has the following methods:

`createElement(tagName)`

This method returns a `Element` object.
The `tagName` parameter is of type **String**.
This method can raise a `DOMException` object.

`createDocumentFragment()`

This method returns a `DocumentFragment` object.

`createTextNode(data)`

This method returns a `Text` object.
The `data` parameter is of type **String**.

`createComment(data)`

This method returns a `Comment` object.
The `data` parameter is of type **String**.

`createCDATASection(data)`

This method returns a `CDATASection` object.
The `data` parameter is of type **String**.
This method can raise a `DOMException` object.

`createProcessingInstruction(target, data)`

This method returns a `ProcessingInstruction` object.
The `target` parameter is of type **String**.
The `data` parameter is of type **String**.
This method can raise a `DOMException` object.

`createAttribute(name)`

This method returns a `Attr` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`createEntityReference(name)`

This method returns a `EntityReference` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`getElementsByTagName(tagname)`

This method returns a `NodeList` object.
The `tagname` parameter is of type **String**.

`importNode(importedNode, deep)`

This method returns a `Node` object.
The `importedNode` parameter is a `Node` object.
The `deep` parameter is of type **Boolean**.
This method can raise a `DOMException` object.

`createElementNS(namespaceURI, qualifiedName)`

This method returns a `Element` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
This method can raise a `DOMException` object.

`createAttributeNS(namespaceURI, qualifiedName)`

This method returns a `Attr` object.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
This method can raise a `DOMException` object.

`getElementsByTagNameNS(namespaceURI, localName)`

This method returns a `NodeList` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`getElementById(elementId)`

This method returns a `Element` object.
The `elementId` parameter is of type **String**.

`adoptNode(source)`

This method returns a `Node` object.
The `source` parameter is a `Node` object.
This method can raise a `DOMException` object.

`setBaseURI(baseURI)`

This method has no return value.
The `baseURI` parameter is of type **String**.
This method can raise a `DOMException` object.

The `Node` class has the following constants:

`Node.ELEMENT_NODE`

This constant is of type **Number** and its value is **1**.

`Node.ATTRIBUTE_NODE`

This constant is of type **Number** and its value is **2**.

`Node.TEXT_NODE`

This constant is of type **Number** and its value is **3**.

`Node.CDATA_SECTION_NODE`

This constant is of type **Number** and its value is **4**.

`Node.ENTITY_REFERENCE_NODE`

This constant is of type **Number** and its value is **5**.

`Node.ENTITY_NODE`

This constant is of type **Number** and its value is **6**.

`Node.PROCESSING_INSTRUCTION_NODE`

This constant is of type **Number** and its value is **7**.

`Node.COMMENT_NODE`

This constant is of type **Number** and its value is **8**.

`Node.DOCUMENT_NODE`

This constant is of type **Number** and its value is **9**.

`Node.DOCUMENT_TYPE_NODE`

This constant is of type **Number** and its value is **10**.

`Node.DOCUMENT_FRAGMENT_NODE`

This constant is of type **Number** and its value is **11**.

`Node.NOTATION_NODE`

This constant is of type **Number** and its value is **12**.

`Node.TREE_POSITION_PRECEDING`

This constant is of type **Number** and its value is **0x01**.

`Node.TREE_POSITION_FOLLOWING`

This constant is of type **Number** and its value is **0x02**.

`Node.TREE_POSITION_ANCESTOR`

This constant is of type **Number** and its value is **0x04**.

`Node.TREE_POSITION_DESCENDANT`

This constant is of type **Number** and its value is **0x08**.

`Node.TREE_POSITION_SAME`

This constant is of type **Number** and its value is **0x10**.

`Node.TREE_POSITION_EXACT_SAME`

This constant is of type **Number** and its value is **0x20**.

`Node.TREE_POSITION_DISCONNECTED`

This constant is of type **Number** and its value is **0x00**.

Object `Node`

The `Node` object has the following properties:

`nodeName`

This read-only property is of type **String**.

`nodeValue`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

`nodeType`

This read-only property is of type **Number**.

`parentNode`

This read-only property is a `Node` object.

`childNodes`

This read-only property is a `NodeList` object.

`firstChild`

This read-only property is a `Node` object.

`lastChild`

This read-only property is a `Node` object.

`previousSibling`

This read-only property is a `Node` object.

`nextSibling`

This read-only property is a `Node` object.

`attributes`

This read-only property is a `NamedNodeMap` object.

`ownerDocument`

This read-only property is a `Document` object.

`namespaceURI`

This read-only property is of type **String**.

`prefix`

This property is of type **String** and can raise a `DOMException` object on setting.

`localName`

This read-only property is of type **String**.

`baseURI`

This read-only property is of type **String**.

`textContent`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

Remarque : cette propriété renvoie la totalité des contenus textuels du nœud courant et de ses enfants. Dans Internet Explorer, cette propriété s'appelle `text` *et non* `textContent`.

The `Node` object has the following methods:

`insertBefore(newChild, refChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
The `refChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`replaceChild(newChild, oldChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
The `oldChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeChild(oldChild)`

This method returns a `Node` object.
The `oldChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`appendChild(newChild)`

This method returns a `Node` object.
The `newChild` parameter is a `Node` object.
This method can raise a `DOMException` object.

`hasChildNodes()`

This method returns a **Boolean**.

`cloneNode(deep)`

This method returns a `Node` object.
The `deep` parameter is of type **Boolean**.

`normalize()`

This method has no return value.

`isSupported(feature, version)`

This method returns a **Boolean**.
The `feature` parameter is of type **String**.
The `version` parameter is of type **String**.

`hasAttributes()`

This method returns a **Boolean**.

`compareTreePosition(other)`

This method returns a **Number**.
The `other` parameter is a `Node` object.
This method can raise a `DOMException` object.

`isSameNode(other)`

This method returns a **Boolean**.
The `other` parameter is a `Node` object.

`lookupNamespacePrefix(namespaceURI)`

This method returns a **String**.
The `namespaceURI` parameter is of type **String**.

`lookupNamespaceURI(prefix)`

This method returns a **String**.
The `prefix` parameter is of type **String**.

`normalizeNS()`

This method has no return value.

`isEqualNode(arg, deep)`

This method returns a **Boolean**.
The `arg` parameter is a `Node` object.
The `deep` parameter is of type **Boolean**.

`getInterface(feature)`

This method returns a `Node` object.
The `feature` parameter is of type **String**.

`setUserData(key, data, handler)`

This method returns a `Object` object.
The `key` parameter is of type **String**.
The `data` parameter is a `Object` object.
The `handler` parameter is a `UserDataHandler` object.

`getUserData(key)`

This method returns a `Object` object.
The `key` parameter is of type **String**.

Object `NodeList`

The `NodeList` object has the following properties:

`length`

This read-only property is of type **Number**.

The `NodeList` object has the following methods:

`item(index)`

This method returns a `Node` object.
The `index` parameter is of type **Number**.
Note: This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer `index` is equivalent to invoking the `item` method with that index.

Remarque : cette note apparaît plusieurs fois dans ce document. Elle signifie qu'il est possible de passer le numéro de l'item recherché soit comme argument de la fonction `item()` *entre parenthèses (*`objetNodeList.item(n)`*), soit comme index de l'objet* `nodeList` *entre crochets (*`objetNodeList[n]`*).*

Object `NamedNodeMap`

The `NamedNodeMap` object has the following properties:

`length`

This read-only property is of type **Number**.

The `NamedNodeMap` object has the following methods:

`getNamedItem(name)`

This method returns a `Node` object.
The `name` parameter is of type **String**.

`setNamedItem(arg)`

This method returns a `Node` object.
The `arg` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeNamedItem(name)`

This method returns a `Node` object.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`item(index)`

This method returns a `Node` object.
The `index` parameter is of type **Number**.
Note: This object can also be dereferenced using square bracket notation (e.g. obj[1]). Dereferencing with an integer `index` is equivalent to invoking the `item` method with that index.

`getNamedItemNS(namespaceURI, localName)`

This method returns a `Node` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setNamedItemNS(arg)`

This method returns a `Node` object.
The `arg` parameter is a `Node` object.
This method can raise a `DOMException` object.

`removeNamedItemNS(namespaceURI, localName)`

This method returns a `Node` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.
This method can raise a `DOMException` object.

Object `CharacterData`

`CharacterData` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `CharacterData` object has the following properties:

`data`

This property is of type **String**, can raise a `DOMException` object on setting and can raise a `DOMException` object on retrieval.

`length`

This read-only property is of type **Number**.

The `CharacterData` object has the following methods:

`substringData(offset, count)`

This method returns a **String**.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
This method can raise a `DOMException` object.

`appendData(arg)`

This method has no return value.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

`insertData(offset, arg)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

`deleteData(offset, count)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
This method can raise a `DOMException` object.

`replaceData(offset, count, arg)`

This method has no return value.
The `offset` parameter is of type **Number**.
The `count` parameter is of type **Number**.
The `arg` parameter is of type **String**.
This method can raise a `DOMException` object.

Object `Attr`

`Attr` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Attr` object has the following properties:

`name`

This read-only property is of type **String**.

`specified`

This read-only property is of type **Boolean**.

`value`

This property is of type **String** and can raise a `DOMException` object on setting.

`ownerElement`

This read-only property is a `Element` object.

Object `Element`

`Element` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Element` object has the following properties:

`tagName`

This read-only property is of type **String**.

Remarque : cette propriété n'est valable que pour le DOM HTML. Elle associe le type de balise auquel est rattaché un élément.

The `Element` object has the following methods:

`getAttribute(name)`

This method returns a **String**.
The `name` parameter is of type **String**.

`setAttribute(name, value)`

This method has no return value.
The `name` parameter is of type **String**.
The `value` parameter is of type **String**.
This method can raise a `DOMException` object.

`removeAttribute(name)`

This method has no return value.
The `name` parameter is of type **String**.
This method can raise a `DOMException` object.

`getAttributeNode(name)`

This method returns a `Attr` object.
The `name` parameter is of type **String**.

`setAttributeNode(newAttr)`

This method returns a `Attr` object.
The `newAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`removeAttributeNode(oldAttr)`

This method returns a `Attr` object.
The `oldAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`getElementsByTagName(name)`

This method returns a `NodeList` object.
The `name` parameter is of type **String**.

`getAttributeNS(namespaceURI, localName)`

This method returns a **String**.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setAttributeNS(namespaceURI, qualifiedName, value)`

This method has no return value.
The `namespaceURI` parameter is of type **String**.
The `qualifiedName` parameter is of type **String**.
The `value` parameter is of type **String**.
This method can raise a `DOMException` object.

`removeAttributeNS(namespaceURI, localName)`

This method has no return value.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.
This method can raise a `DOMException` object.

`getAttributeNodeNS(namespaceURI, localName)`

This method returns a `Attr` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`setAttributeNodeNS(newAttr)`

This method returns a `Attr` object.
The `newAttr` parameter is a `Attr` object.
This method can raise a `DOMException` object.

`getElementsByTagNameNS(namespaceURI, localName)`

This method returns a `NodeList` object.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

`hasAttribute(name)`

This method returns a **Boolean**.
The `name` parameter is of type **String**.

`hasAttributeNS(namespaceURI, localName)`

This method returns a **Boolean**.
The `namespaceURI` parameter is of type **String**.
The `localName` parameter is of type **String**.

Object `Text`

`Text` has all the properties and methods of the `CharacterData` object as well as the properties and methods defined below.

The `Text` object has the following properties:

`isWhitespaceInElementContent`

This read-only property is of type **Boolean**.

`wholeText`

This read-only property is of type **String**.

The `Text` object has the following methods:

`splitText(offset)`

This method returns a `Text` object.
The `offset` parameter is of type **Number**.
This method can raise a `DOMException` object.

`replaceWholeText(content)`

This method returns a `Text` object.
The `content` parameter is of type **String**.
This method can raise a `DOMException` object.

Object `Comment`

`Comment` has all the properties and methods of the `CharacterData` object.

Prototype Object `UserDataHandler`

The `UserDataHandler` class has the following constants:

`UserDataHandler.CLONED`

This constant is of type **Number** and its value is **1**.

`UserDataHandler.IMPORTED`

This constant is of type **Number** and its value is **2**.

`UserDataHandler.DELETED`

This constant is of type **Number** and its value is **3**.

Object `UserDataHandler`

The `UserDataHandler` object has the following methods:

`handle(operation, key, data, src, dst)`

This method has no return value.
The `operation` parameter is of type **Number**.
The `key` parameter is of type **String**.
The `data` parameter is a `Object` object.
The `src` parameter is a `Node` object.
The `dst` parameter is a `Node` object.

Prototype **`Object`** `DOMError`

The `DOMError` class has the following constants:

`DOMError.SEVERITY_WARNING`

This constant is of type **Number** and its value is **0**.

`DOMError.SEVERITY_ERROR`

This constant is of type **Number** and its value is **1**.

`DOMError.SEVERITY_FATAL_ERROR`

This constant is of type **Number** and its value is **2**.

Object `DOMError`

The `DOMError` object has the following properties:

`severity`

This read-only property is of type **Number**.

`message`

This read-only property is of type **String**.

`exception`

This read-only property is a `Object` object.

`location`

This read-only property is a `DOMLocator` object.

Object `DOMErrorHandler`

The `DOMErrorHandler` object has the following methods:

`handleError(error)`

This method returns a **Boolean**.
The `error` parameter is a `DOMError` object.

Object `DOMLocator`

The `DOMLocator` object has the following properties:

`lineNumber`

This read-only property is of type **Number**.

`columnNumber`

This read-only property is of type **Number**.

`offset`

This read-only property is of type **Number**.

`errorNode`

This read-only property is a `Node` object.

`uri`

This read-only property is of type **String**.

Object `CDATASection`

`CDATASection` has all the properties and methods of the `Text` object.

Object `DocumentType`

`DocumentType` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `DocumentType` object has the following properties:

`name`

This read-only property is of type **String**.

`entities`

This read-only property is a `NamedNodeMap` object.

`notations`

This read-only property is a `NamedNodeMap` object.

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

`internalSubset`

This read-only property is of type **String**.

Object `Notation`

`Notation` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Notation` object has the following properties:

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

Object `Entity`

`Entity` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `Entity` object has the following properties:

`publicId`

This read-only property is of type **String**.

`systemId`

This read-only property is of type **String**.

`notationName`

This read-only property is of type **String**.

`actualEncoding`

This property is of type **String**.

`encoding`

This property is of type **String**.

`version`

This property is of type **String**.

Object `EntityReference`

`EntityReference` has all the properties and methods of the `Node` object.

Object `ProcessingInstruction`

`ProcessingInstruction` has all the properties and methods of the `Node` object as well as the properties and methods defined below.

The `ProcessingInstruction` object has the following properties:

`target`

This read-only property is of type **String**.

`data`

This property is of type **String** and can raise a `DOMException` object on setting.

3.2.4.2 Structuration des interfaces

Le DOM présente tout document comme une hiérarchie d'objets de type `Node`. Ces nœuds implémentent différentes interfaces, qui peuvent chacune avoir différents types de nœuds enfants. La liste suivante décrit les relations de filiation autorisées (extraite de DOM Level 3).

`Document` `Element` (maximum : 1),
`ProcessingInstruction`,
`Comment`,
`DocumentType` (maximum : 1)

`DocumentFragment`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`DocumentType`	pas d'enfant
`EntityReference`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`Element`	`Attr`, `Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDATASection`, `EntityReference`
`Attr`	`Text`, `EntityReference`
`ProcessingInstruction`	pas d'enfant
`Comment`	pas d'enfant
`Text`	pas d'enfant

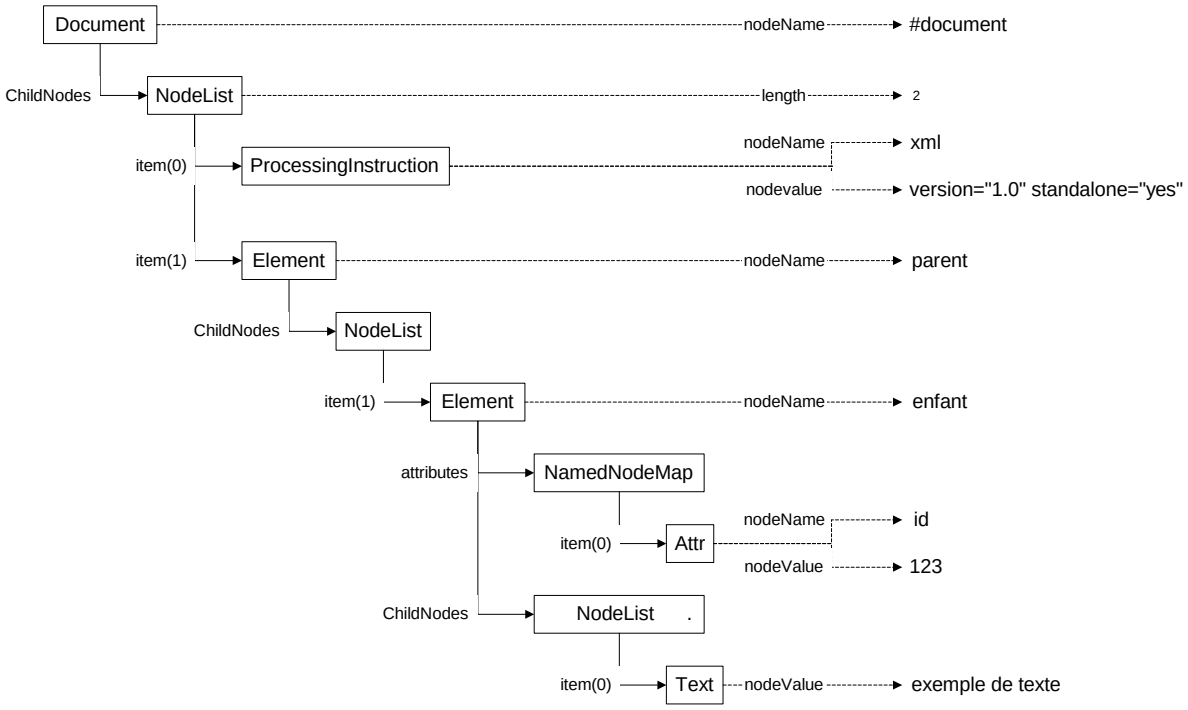
`CDATASection`	pas d'enfant
`Entity`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`Notation`	pas d'enfant

3.2.5 Exemple de représentation d'un document XML par le DOM

Considérons le document XML suivant :

```
<?xml version="1.0" standalone="yes"?>
<parent>
     <enfant id="123">exemple de texte</enfant>
</parent>
```

Le schéma suivant montre l'arborescence de nœuds qui permet de représenter ce document avec le DOM.



Légende : les rectangles représentent les nœuds du modèle objet associé au document par le DOM. Les textes à l'intérieur de ces rectangles indiquent les types des nœuds (correspondant aux constantes accessibles par la propriété `nodeType`). Les propriétés permettant d'accéder à ces nœuds sont indiquées par les textes à gauche des flèches. Le contenu de ces nœuds est indiqué à droite, ainsi que les propriétés qui permettent d'y accéder.

Remarques :

La valeur de la propriété `nodeName` appliquée au nœud racine du document est empirique. Elle est attribuée à ce nœud par Internet Explorer 5.0.

Comme décrit au paragraphe précédent, il est possible de « passer » directement du nœud `Document` au premier nœud `Element` par la propriété `documentElement`.

Il est possible d'aller plus loin dans la décomposition de l'interface `ProcessingInstruction`. Les différentes parties de la déclaration XML peuvent également être obtenues séparément via des attributs de cette interface.

`NodeList` et `NamedNodeMap` sont des interfaces « *vivantes* » (*live*). Cela signifie que toutes les manipulations de la structure du document sont immédiatement répercutées sur les interfaces de ce type. Par exemple, l'ajout d'un élément fils à un élément existant provoque l'apparition du nœud correspondant dans l'interface `NodeList` relative à ce nœud.

Sur ce schéma, on voit que pour obtenir le contenu textuel d'un nœud, il faut utiliser la propriété `nodeValue` du nœud `Text` associé. Avec Internet Explorer®, un tel nœud a pour nom (propriété `nodeName`) : "`#text`".

3.2.6 Utilisation du DOM avec MSXML

Ce paragraphe donne un aperçu plus pratique de l'utilisation qui peut être faite du DOM pour manipuler un document XML avec Internet Explorer® (version 5.0 et suivantes). Il présente quelques opérations de base permettant d'utiliser le DOM, notamment dans des fonctions JScript.

3.2.6.1 Chargement du fichier

La première chose à faire est de déclarer le document accédé par le DOM dans une variable (ou un objet) JScript. Internet Explorer® utilise la technologie ActiveX (anciennement OLE) pour traiter ces objets. Comme il s'agit d'un document du DOM accédé par le parser XML, la syntaxe est :

```
var XMLDoc = new ActiveXObject("MSXML.DOMDocument");
XMLDoc.async=false;
```

Remarques :

Le mot-clé `var` est optionnel dans la déclaration ci-dessus.

La seconde ligne évite le chargement du document en tâche de fond. L'accès au document via le DOM n'étant possible que lorsque celui-ci est entièrement chargé, cela dispense d'employer une temporisation (`window.setTimeout()`) et un test de la propriété `readyState` du contrôle ActiveX[32]. L'inconvénient est la perte de temps que peut occasionner le chargement d'un document volumineux.

[32] Cette propriété renvoie un entier positif entre 0 et 3 tant que le document n'est pas totalement chargé et 4 ensuite.

Il est possible de spécifier si l'on souhaite ou non que le document soit validé (par exemple par rapport à une DTD). Pour cela, il faut utiliser la propriété booléenne `validateOnParse` de l'objet `XMLDoc`. La valeur par défaut de cette propriété est `true` lorsque l'on accède à un document XML par le DOM (alors qu'elle est `false` lorsque l'on affiche directement ce document dans Internet Explorer®. Cette méthode ne permet pas la validation par les schémas XML. Celle-ci n'est pas encore correctement implémentée dans Internet Explorer® et n'est pas abordée ici.

Il faut ensuite charger le fichier dans cet objet. Deux méthodes peuvent être utilisées : `load()` et `loadXML()`. L'exemple suivant utilise la première, qui permet l'utilisation de noms de fichiers et de chemins relatifs.

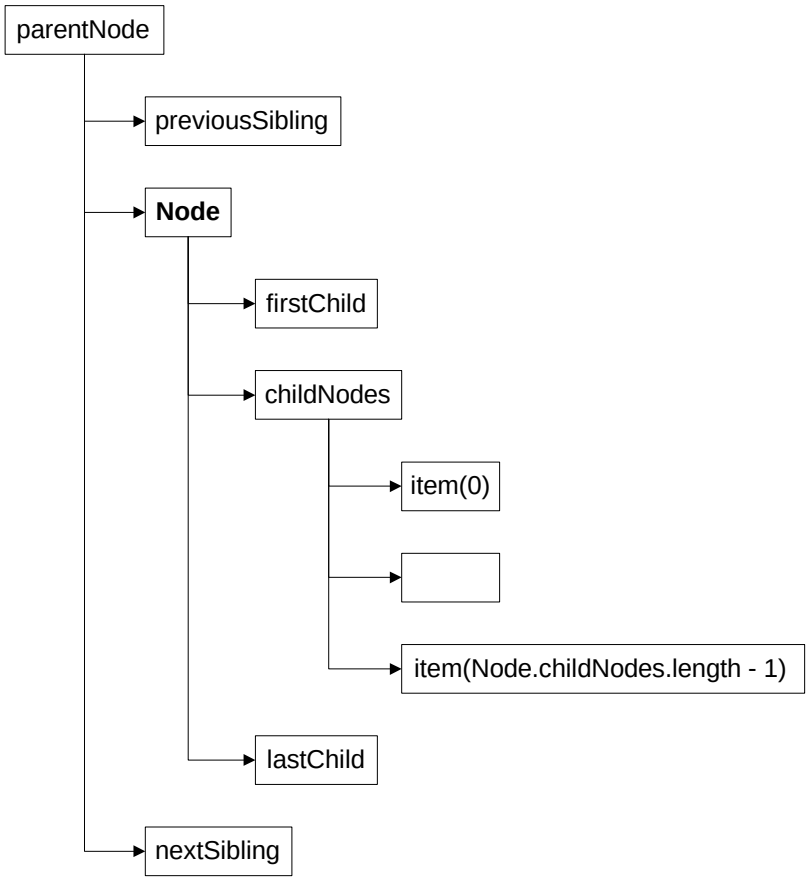
```
XMLDoc.load("fichier.xml");
```

L'objet `XMLDoc` obtenu est alors une instance de l'interface `Document` du DOM.

3.2.6.2 Accès à l'élément racine

La propriété `documentElement` d'un objet `Document` (ici, `XMLDoc`) donne accès à l'élément racine du document XML. Cette propriété renvoie un objet `Element`, qui est un type particulier de `Node`. L'élément racine peut alors être traité comme tous les nœuds de l'arborescence, notamment pour accéder aux autres éléments du document.

3.2.6.3 Déplacement dans l'arborescence des éléments



Le schéma ci-contre est issu des spécifications du DOM du W3C. Il indique les propriétés de l'interface `Node` permettant, depuis un nœud représentant un élément du document, d'accéder aux autres éléments. Ces propriétés permettent d'accéder aux nœuds représentant des éléments XML ou HTML. Sur ce schéma, on suppose être positionné sur l'interface `Node` indiquée en gras.

Remarques : le déplacement se fait selon l'arborescence du modèle que le DOM renvoie du document XML, qui peut différer sensiblement de l'arborescence du

document lui-même. Par exemple, dans le DOM, les attributs sont représentés par des nœuds enfants du nœud représentant l'élément du document auquel ils se rapportent. Un exemple de cette différence de structures est donné au paragraphe suivant.

3.2.7 Utilisation du DOM en Java

Côté Java, nous avons vu plus haut que JAXP fournit un package org.xml.dom. Nous présentons ci-dessous les différentes interfaces implémentables, ainsi que l'exception qui peut être levée lors du traitement d'un document XML par un parser DOM (tiré de http://java.sun.com/j2se/1.4.2/docs/api) :

Interfaces

`Attr`	The Attr interface represents an attribute in an Element object.
`CDATASection`	CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup.
`CharacterData`	The CharacterData interface extends Node with a set of attributes and methods for accessing character data in the DOM.
`Comment`	This interface inherits from CharacterData and represents the content of a comment, i.e., all the characters between the starting ' <!--' and ending '-->'.
`Document`	The Document interface represents the entire HTML or XML document.
`DocumentFragment`	DocumentFragment is a "lightweight" or "minimal" Document object.
`DocumentType`	Each Document has a doctype attribute whose value is either null or a DocumentType object.
`DOMImplementation`	The DOMImplementation interface provides a number of methods for performing operations that are independent of any particular instance of the document object model.
`Element`	The Element interface represents an element in an HTML or XML document.
`Entity`	This interface represents an entity, either parsed or unparsed, in an XML document.
`EntityReference`	EntityReference objects may be inserted into the structure model when an entity reference is in the source document, or when the user wishes to insert an entity reference.
`NamedNodeMap`	Objects implementing the NamedNodeMap interface are used to represent collections of nodes that can be accessed by name.
`Node`	The Node interface is the primary datatype for the entire Document Object Model.
`NodeList`	The NodeList interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented.
`Notation`	This interface represents a notation declared in the DTD.
`ProcessingInstruction`	The ProcessingInstruction interface represents a "processing instruction", used in XML as a way to keep processor-specific information in the text of the document.
`Text`	The Text interface inherits from CharacterData and represents the textual content (termed character data in XML) of an

	Element or Attr.

Exception

`DOMException`	DOM operations only raise exceptions in "exceptional" circumstances, i.e., when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable).

Nous n'entrerons pas ici dans le détail de chacune de ces interfaces et exceptions, dans la mesure où elles possèdent des méthodes très similaires à celles décrites plus haut.

3.3 SAX (Simple API for XML)

SAX est une interface de programmation pour applications développée hors du cadre du W3C par une communauté d'utilisateurs-développeurs de la technologie XML[33]. SAX a été conçue comme une alternative au DOM, qui se fonde sur la programmation événementielle[34] pour traiter le parsing de documents XML. Il n'existe donc pas de recommandation SAX, mais des spécifications, disponibles en ligne sur le site : http://www.saxproject.org. SAX a été originellement écrite pour le langage Java, mais cette API a été ensuite déclinée dans d'autres langages parmi lesquels C++, Perl, Python ou Pascal. Dans ce cours, nous étudions les versions 1 et 2 de SAX pou r Java.

Avertissement : la spécification de SAX fournit deux types d'indications, destinées soit aux développeurs qui implémentent des parsers compatibles SAX, soit à ceux qui développent des applications simples utilisant de tels parsers. Dans ce cours, nous ne nous intéressons qu'à la seconde catégorie. La description qui suit est donc intentionnellement incomplète, et ne permet de construire que des applications simples. Dès que le niveau de technicité augmente, il devient indispensable de s'intéresser aux autres interfaces non mentionnées ici.

Le principe général de SAX est de fournir des interfaces et des classes pour la gestion d'événements SAX, qui devront être dérivées dans les applications traitant des données XML. Ces interfaces sont regroupées dans les packages `org.xml.sax`, `org.xml.sax.helpers` et `org.xml.sax.ext`. Le premier fournit les éléments essentiels de la distribution SAX (« the core SAX distribution »), tandis que le deuxième fournit des classes et des interfaces qui ne sont pas nécessaires à la distribution, mais sont uniquement destinées à faciliter le travail des développeurs. Celles-ci font référence au contenu du premier package et ne sont pas nécessairement distribuées dans d'autres langages que Java. Le troisième et dernier package est une extension du premier, et fournit deux interfaces supplémentaires optionnelles pour la gestion de certains types d'événements[35] qui ne sont pas abordés ici.

---

[33] Cette communauté est organisée autour de la liste de diffusion « xml-dev», hébergée par http://www.xml.org.

[34] Pour résumer le concept de programmation événementielle en Java, nous dirons que cela consiste en un simple appel de fonctions prédéterminées, membres d'une classe qui dérive d'une classe abstraite ou d'une interface prévues à cet effet. Il ne s'agit pas d'un mécanisme interne au langage comme la gestion des exceptions ou des erreurs.

[35] Événements de déclaration de DTD à l'intérieur d'un document XML et lexicaux (comme les sections CDATA ou les commentaires).

3.3.1 Le package `org.xml.sax`

Le package `org.xml.sax` a subi plusieurs modifications entre les versions 1.0 (cf l'exemple de code au début de ce chapitre) et 2.0. Ci-dessous sa description telle qu'elle est donnée en version 1.0, et les modifications apportées par la version 2.

3.3.1.1 Les interfaces SAX standard

SAX définit les interfaces suivantes, qui peuvent[36] être implémentées par les applications utilisant cette API. Les applications les plus simples peuvent ne nécessiter que DocumentHandler ou peut-être ErrorHandler. Une application peut implémenter toutes ces interfaces dans la même classe. Le contenu de ce paragraphe est tiré de l'« overview » de SAX 1.0 présentée à l'URL : http://www.saxproject.org/?selected=sax1.

`DocumentHandler`

Ceci est l'interface la plus importante de SAX 1.0. C'est celle qui reçoit la notification des principaux événements relatifs au traitement d'un élément XML, tels que les balises de début de document ou d'éléments, de fin d'éléments ou de document, les instructions de traitement etc. Cette méthode est dépréciée en version 2.0, et remplacée par la suivante :

`ContentHandler`

Cette interface SAX 2.0, similaire à la précédente apporte essentiellement un meilleur support aux espaces de noms et la prise en compte des entités non analysées (« skipped entities »)[37] par rapport à celle-ci. Les méthodes abstraites de cette interface sont les suivantes :

- `void characters(char[] ch, int start, int length)` : Receive notification of character data.
- `void endDocument()` : Receive notification of the end of a document.
- `void endElement(String namespaceURI, String localName, String qName)` : Receive notification of the end of an element.
- `void endPrefixMapping(String prefix)` : End the scope of a prefix-URI mapping.
- `void ignorableWhitespace(char[] ch, int start, int length)` : Receive notification of ignorable whitespace in element content.
- `void processingInstruction(String target, String data)` : Receive notification of a processing instruction.
- `void setDocumentLocator(Locator locator)` : Receive an object for locating the origin of SAX document events.
- `void skippedEntity(String name)` : Receive notification of a skipped entity.
- `void startDocument()` : Receive notification of the beginning of a document.

---

[36] « A SAX application may implement any or none of the following interfaces. »

[37] Rappel : lorsqu'une référence à une entité définie à l'extérieur d'un document XML est rencontrée par un parser non validant, celui-ci ne peut vérifier qu'elle est effectivement définie, et « saute » donc cette référence. D'où le nom de « skipped entities » en Anglais.

- `void startElement(String namespaceURI, String localName, String qName, Attributes atts)` : Receive notification of the beginning of an element.
- `void startPrefixMapping(String prefix, String uri)` : Begin the scope of a prefix-URI Namespace mapping.

`ErrorHandler`

Toute application qui utilise une gestion d'erreurs spécifique doit implémenter cette interface et la déclarer la classe correspondante en tant que gestionnaire d'erreurs au parser via la méthode XMLReader.setErrorHandler(ErrorHandler handler). Cette interface possède trois méthodes :

- `void error(SAXParseException exception)` : Receive notification a recoverable error.
- `void fatalError(SAXParseException exception)` : Receive notification an unrecoverable error.
- `void warning(SAXParseException exception)` : Receive notification a warning.

`DTDHandler`

Toute application qui travaille avec des notations ou des entités non analysées doit implémenter cette interface pour recevoir la notification des déclarations NOTATION et ENTITY dans une DTD. Cette interface possède deux méthodes :

- `void notationDecl(String name, String publicId, String systemId)` : Receive notification of a notation declaration event.
- `void unparsedEntityDecl(String name, String publicId, String systemId, String notationName)` : Receive notification an unparsed entity declaration event.

`EntityResolver`

Toute application qui a besoin de rediriger des URIs dans les documents (ou effectuer d'autres types de getion spécifique des entités), doit fournir une implémentation de cette interface. Cette interface possède une méthode :

- `InputSource resolveEntity(String publicId, String systemId)` : Allow the application to resolve external entities.

`Attributes`

Cette interface est spécifique à SAX 2.0. Elle remplace l'interface `AttributeList` de SAX 1.0, en fournissant une meilleure gestion des espaces de noms. Elle définit le type de données abstrait nécessaire pour implémenter la méthode `startElement` de `ContentHandler`. Il n'est pas nécessaire d'implémenter cette méthode dans une application, car une implémentation est fournie par la classe `AttributesImpl` du package `org.xml.sax.helpers`.

`XMLReader`

L'interface `XMLReader` représente un parser XML satisfaisant SAX 2.0 et n'a pas à être implémentée dans une application. Nous la signalons toutefois ici, car elle succède à l'interface `Parser` de SAX 1.0, en améliorant, comme d'habitude, la gestion des espaces de noms. Cette succession implique des changements dans la méthode de création d'une instance de parser avec SAX 2.0 : une instance de parser en SAX 2.0 peut être créée soit avec la méthode `SAXParserFactory.newSAXParser()` du package `javax.xml.parsers` (qui instanciait une implémentation de `Parser` en SAX 1.0), soit avec la méthode `XMLReaderFactory.createXMLReader()` du package `org.xml.sax.helpers` (spécifique à SAX 2.0). Toute instance de classe implémentant `XMLReader` possède les méthodes suivantes :

- `ContentHandler getContentHandler()` : Return the current content handler.
- `DTDHandler getDTDHandler()` : Return the current DTD handler.
- `EntityResolver getEntityResolver()` : Return the current entity resolver.
- `ErrorHandler getErrorHandler()` : Return the current error handler.
- `Boolean getFeature(String name)` : Look up the value of a feature.
- `Object getProperty(String name)` : Look up the value of a property.
- `Void parse(InputSource input)` : Parse an XML document.
- `Void parse(String systemId)` : Parse an XML document from a system identifier (URI).
- `Void setContentHandler(ContentHandler handler)` : Allow an application to register a content event handler.
- `Void setDTDHandler(DTDHandler handler)` : Allow an application to register a DTD event handler.
- `Void setEntityResolver(EntityResolver resolver)` : Allow an application to register an entity resolver.
- `Void setErrorHandler(ErrorHandler handler)` : Allow an application to register an error event handler.
- `Void setFeature(String name, boolean value)` : Set the state of a feature.
- `Void setProperty(String name, Object value)` : Set the value of a property.

Remarque : dans la plupart des cas, vous n'aurez pas besoin d'utiliser d'autre méthode que les méthodes `parse()` et éventuellement `setErrorHandler()`.

3.3.1.2 Les classes SAX standard

SAX 1.0 fournit deux classes destinées à être dérivée (pour la première) ou instanciée (pour la seconde) pour faciliter l'implémentation d'applications.

`HandlerBase`

Cette classe fournit une implémentation par défaut de l'ensemble des interfaces `DocumentHandler`, `ErrorHandler`, `DTDHandler` et `EntityResolver` de SAX 1.0. Cette classe est très utile pour la programmation d'applications, car elle permet de ne réécrire que les méthodes spécifiques à son application. Comme l'interface `DocumentHandler` n'est

plaus valable en SAX 2.0, cette classe est également dépréciée. Elle est remplacée par la classe `ContentHandler`, qui se trouve désormais dans le package `org.xml.sax.helpers` (ce qui semble d'ailleurs plus logique…).

`InputSource`

Cette classe contient toutes les informations nécessaires pour instancier une source d'entrée (« input source ») simple, parmi lesquelles ses différents types d'identifiants (« public identifier », « system identifier »), les types de flux dont elle est issue (« byte stream », ou « character stream »). Toute application doit instancier au moins une InputSource pour le parser. L'interface `EntityResolver` possède une méthode `resolveEntity()` pour résoudre les entités externes d'un document qui renvoie un objet de type InputSource.

3.3.1.3 Les exceptions SAX

SAX 1.0 fournit deux classes de gestion d'exceptions et deux autres sont définies par la spécification SAX 2.0. Elles sont listées selon cet ordre. Ces classes s'utilisent comme des classes de gestion d'exception standard, dans un bloc `try {...} catch {...}`.

`SAXException`

Cette classe représente une exception SAX quelconque. Les trois exceptions suivantes dérivent de celle-ci. Elle possède trois méthodes :

- `Exception getException()` : Return the embedded exception, if any.
- `String getMessage()` : Return a detail message for this exception.
- `String toString()` : Override toString to pick up any embedded exception.

`SAXParseException`

Cette classe représente une exception SAX liée à un point spécifique d'un document XML source. Elle possède les méthodes de la classe précédente plus les suivantes :

- `int getColumnNumber()` : The column number of the end of the text where the exception occurred.
- `int getLineNumber()` : The line number of the end of the text where the exception occurred.
- `String getPublicId()` : Get the public identifier of the entity where the exception occurred.
- `String getSystemId()` : Get the system identifier of the entity where the exception occurred.

`SAXNotRecognizedException`

Cette classe représente une exception SAX 2.0 liée à la non identification d'une propriété ou d'une donnée quelconque dans un document XML. Elle ne possède pas de méthode spécifique.

`SAXNotSupportedException`

Cette classe représente une exception SAX 2.0 liée à une opération non supportée par une propriété ou une donnée identifiée d'un document. Elle ne possède pas de méthode spécifique.

3.3.2 Le package `org.xml.sax.helpers`

Ce package ne fournit que des classes totalement implémentées, qui ont pour but de faciliter la programmation d'applications utilisant SAX. L'objectif est de créer des instances de ces classes, en en redéfinissant au besoin certaines méthodes. Nous n'évoquerons ici que certaines de ces classes. Comme précédemment, la totalité du package est détaillée à l'URL : http://java.sun.com/j2se/1.4.2/docs/api/index.html.

`AttributesImpl` (SAX 2.0), qui fait suite à `AttributeListImpl` (SAX 1.0), fournit, comme son nom l'indique, une implémentation de l'interface `Attributes`. Elle possède les méthodes suivantes :

- `void addAttribute(String uri, String localName, String qName, String type, String value)` : Add an attribute to the end of the list.
- `void clear()` : Clear the attribute list for reuse.
- `int getIndex(String qName)` : Look up an attribute's index by qualified (prefixed) name.
- `int getIndex(String uri, String localName)` : Look up an attribute's index by Namespace name.
- `int getLength()` : Return the number of attributes in the list.
- `String getLocalName(int index)` : Return an attribute's local name.
- `String getQName(int index)` : Return an attribute's qualified (prefixed) name.
- `String getType(int index)` : Return an attribute's type by index.
- `String getType(String qName)` : Look up an attribute's type by qualified (prefixed) name.
- `String getType(String uri, String localName)` : Look up an attribute's type by Namespace-qualified name.
- `String getURI(int index)` : Return an attribute's Namespace URI.
- `String getValue(int index)` : Return an attribute's value by index.
- `String getValue(String qName)` : Look up an attribute's value by qualified (prefixed) name.
- `String getValue(String uri, String localName)` : Look up an attribute's value by Namespace-qualified name.
- `void removeAttribute(int index)` : Remove an attribute from the list.
- `void setAttribute(int index, String uri, String localName, String qName, String type, String value)` : Set an attribute in the list.
- `void setAttributes(Attributes atts)` : Copy an entire Attributes object.
- `void setLocalName(int index, String localName)` : Set the local name of a specific attribute.
- `void setQName(int index, String qName)` : Set the qualified name of a specific attribute.
- `void setType(int index, String type)` : Set the type of a specific attribute.
- `void setURI(int index, String uri)` : Set the Namespace URI of a specific attribute.

- `void setValue(int index, String value)` : Set the value of a specific attribute.

`DefaultHandler` (SAX 2.0), qui fait suite à `DocumentHandler` (package `org.xml.sax`, SAX 1.0), implémente les interfaces `EntityResolver`, `DTDHandler`, `ContentHandler` et `ErrorHandler`. Elle possède les méthodes de toutes ces interfaces.

`XMLReaderAdapter` permet de faire passer (« wrapper ») un parser de type `XMLReader` pour une implémentation de l'interface `Parser` de SAX 1.0. Cela peut être utile quand votre application utilise SAX 2.0 mais votre parser n'implémente que la version 1.0 de l'API. Cette classe implémente à la fois les interfaces `Parser` de SAX 1.0 et `ContentHandler` de SAX 2.0.

`XMLReaderFactory` (SAX 2.0), qui fait suite à `ParserFactory` (SAX 1.0), permet de créer une instance de parser `XMLReader` SAX 2.0. Pour cela, il suffit de déclarer une instance de `XMLReaderFactory`, et d'appeler sa méthode `createXMLReader()`. Cette classe possède deux méthodes :

- `static XMLReader createXMLReader()` : Attempt to create an XML reader from a system property.
- `static XMLReader createXMLReader(String className)` : Attempt to create an XML reader from a class name.