Alexandre Brillant

XML

Cours et exercices

Modélisation • Schémas et DTD • Design patterns • XSLT • DOM • RelaxNG • XPath • SOAP • XQuery • XSL-FO • SVG

EYROLLES

XML

Cours et exercices

CHEZ LE MÊME ÉDITEUR

Ouvrages sur le même sujet

Ph. Drix. – **XSLT fondamental.** *Avec 20 design patterns prêts à l'emploi.*
N°11082, 2002, 500 pages.

A. Lonjon, J.-J. Thomasson. – **Modélisation XML.**
N°11521, 2006, 498 pages (collection Architecte logiciel).

J. Protzenko, B. Picaud. – **XUL.** N°11675, 2005, 320 pages.

C. Porteneuve, préface de T. Nitot – **Bien développer pour le Web 2.0 – Bonnes pratiques Ajax.**
N°12028, 2007, 580 pages.

S. Crozat. – **Scenari – La chaîne éditoriale libre.** N°12150, 2007, 200 pages.

R. Fleury – **Java/XML.** N°11316, 2004, 228 pages.

J.-J. Thomasson. – **Schémas XML**. N°11195, 2002, 500 pages.

L. Maesano, C. Bernard, X. Legalles. – **Services Web en J2EE et .Net.**
N°11067, 2003, 1088 pages.

Dans la même collection

H. Bersini, I. Wellesz. – **L'orienté objet**. *Cours et exercices en UML 2 avec PHP, Java, Python, C# et C++*
N°12084, 3e édition 2007, 520 pages (collection Noire).

X Blanc, I. Mounier. – **UML 2 pour les développeurs.**
N°12029, 2006, 202 pages.

A. Tasso. – **Le livre de Java premier langage.** N°11994, 4e édition 2006, 472 pages, avec CD-Rom.

P. Roques. – **UML 2 par la pratique.** N°12014, 5e édition 2006, 385 pages.

Autres ouvrages

E. Sloïm. – **Sites web.** *Les bonnes pratiques.*
N°12101, 2007, 14 pages.

R. Rimelé. – **Mémento MySQL.** N°12012, 2007, 14 pages.

C. Pierre de Geyer et G. Ponçon. – **Mémento PHP et SQL.** N°11785, 2006, 14 pages.

M. Grey. – **Mémento Firefox et Thunderbird** N°11780, 2006, 14 pages.

R. Goetter. – **CSS 2 : pratique du design web** . N°11976, 2e édition 2007, 324 pages.

I. Jacobson, G. Booch, J.Rumbaugh. – **Le Processus unifié de développement logiciel**.
N°9142, 2000, 487 pages.

P. Rigaux, A. Rochfeld. – **Traité de modélisation objet**. N°11035, 2002, 308 pages.

B. Meyer. – **Conception et programmation orientées objet**. N°9111, 2000, 1223 pages.

Alexandre Brillant

XML

Cours et exercices

Modélisation - Schéma - Design patterns - XSLT - XPath - SOAP - XQuery - XSL-FO – SVG

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Remerciements à Jean-Marie Gouarné pour les précisions sur les formats OpenOffice.org et OpenXML ainsi qu'à Stéphane Crozat pour les informations concernant la chaîne éditoriale XML Scenari.

Table des matières

Remerciements

Je tiens à remercier toutes les personnes qui ont participé à cet ouvrage :

- les relecteurs et correcteurs des éditions Eyrolles (Sophie Hincelin, Hind Boughedaoui et Eric Bernauer),
- Jean-Marie Gouarné pour ses précisions sur les formats Open XML et Open Document,
- Stéphane Crozat, pour son aparté sur la chaîne éditoriale libre Scenari,
- mon éditrice, Muriel Shan Sei Fan, pour ses propositions d'amélioration.

Avant-propos

XML : Une galaxie en formation

XML se démocratise, à tel point qu'il en deviendrait presque encombrant dans la mesure où son utilisation n'est pas toujours justifiée. Les maîtres mots deviennent flexibilité et ouverture. La hantise des formats fermés, nécessitant des opérations coûteuses de traduction lorsque les informations circulent en dehors de leur cadre classique d'exploitation, ont donné à XML un rôle majeur dans l'activité informatique. Mais XML cache en réalité une grande diversité d'utilisations et de fonctionnement. Les langages à connaître sont variés et s'emboîtent les uns dans les autres. XML, pourtant voué à la simplicité des échanges, est de ce fait devenu, à force d'enrichissements, un ensemble conséquent qu'il n'est pas forcément aisé de maîtriser.

L'objectif de ce livre

Cet ouvrage a pour objectif premier de vous aider à comprendre les technologies XML. Seuls les points de détail sont omis car déjà disponibles dans les documents de spécification. L'ouvrage tente malgré tout d'éviter le défaut trop fréquent de n'aborder XML qu'en surface. Le but de ce livre est donc de vous faire comprendre la mécanique XML, de vous donner tous les instruments pour travailler efficacement en ne perdant pas de temps à cerner un jargon parfois inutilement complexe. En un mot : aller à l'essentiel pour être productif.

À qui s'adresse cet ouvrage ?

Cet ouvrage s'adresse avant tout à des personnes qui sont ou seront amenées à manipuler des documents XML, qu'ils soient étudiants, enseignants, développeurs, architectes ou chefs de projets. Cependant, si vous n'êtes pas familiarisé avec la programmation, cet ouvrage reste néanmoins abordable puisqu'il propose de nombreux exemples et rappels distillés au fil du texte. Lorsqu'un point de vocabulaire fait référence à des éléments de programmation, une explication concise l'accompagne pour en faciliter la compréhension.

Structure de l'ouvrage

Tout au long de cet ouvrage, notamment destiné à des étudiants et à leurs enseignants, l'exposé laisse place à des exercices et leur solution, pour que le lecteur approfondisse les technologies qui l'intéressent.

Le **chapitre 1** fait le point sur l'intégration de XML dans les entreprises. En quelques paragraphes les enjeux et l'intégration logique de XML sont développés.

Le **chapitre 2** est incontournable pour comprendre un document XML puisqu'il explique la structure de base ainsi que la notion d'espace de noms. Il est indispensable d'étudier ce chapitre avant d'aborder les suivants.

Le **chapitre 3** traite de la validation par l'intermédiaire des DTD et des schémas W3C. Cette notion de validation est fondamentale puisqu'elle sert à conforter la cohérence des structures que l'on peut mettre en place. On pourrait la comparer au plan d'un architecte.

Le **chapitre 4** approfondit l'utilisation des schémas W3C et propose des techniques de modélisation. Ce chapitre est peut-être le plus complexe et nécessite une bonne compréhension du chapitre précédent.

Le **chapitre 5** concerne la publication de vos documents. Vous y apprendrez à rendre vos documents XML visibles, par exemple, sous forme de pages HTML ou de documents PDF.

Le **chapitre 6** est davantage réservé à des développeurs puisqu'il s'intéresse particulièrement à la communication inter-applications avec XML, notamment par le biais des services web. Il peut également servir à des administrateurs qui ont besoin de connaître la nature des échanges applicatifs.

Le **chapitre 7** vous initie à l'intégration de XML aux bases de données. Il s'agit aussi bien des bases traditionnelles de type relationnelles que des bases dites pures XML. Dans cette dernière catégorie, deux exemples de base vous seront proposés avec d'une part des manipulations en ligne de commande réalisables directement, et d'autre part des manipulations par programmation destinées aux développeurs.

Le **chapitre 8** est quant à lui destiné aux seuls développeurs puisqu'il traite de la programmation avec XML. Les modèles dits classiques, à savoir SAX et DOM, ont été étudiés avec différents langages. D'autres formes de programmation plus efficaces ont également été analysées, comme les systèmes par *mapping* offrant une programmation XML presque transparente.

1

Le document XML

L'objectif de ce premier chapitre est de vous guider dans l'intégration du formalisme XML dans les entreprises. Ce dernier est né d'un besoin universel : savoir faire cohabiter dans un même document de l'information et de la signification. D'une manière informelle, un document XML peut être perçu comme un document texte porteur de ces deux types de données.

Rôle du document XML

L'entreprise fournit des services dont la production nécessite généralement plusieurs étapes. À chaque étape, des informations peuvent être produites et/ou consommées. Le rôle de l'informatique est d'offrir un cadre de stockage et de traitement de l'ensemble de ces informations. Pour être comprise, toute information doit être formalisée, c'est-à-dire représentée en respectant certaines règles. Le choix des mots, l'ordre des mots, etc., tout cela a du sens pour les acteurs de l'entreprise, qu'ils soient humains ou logiciels. Un document XML sert alors de vecteur à l'information : c'est une manière universelle de représenter des données et leur sens dans un cadre précis.

Considérons l'exemple d'une entreprise, organisée en différents services, qui demande à un cabinet externe de réaliser des bilans de son activité. Ces bilans peuvent influencer le fonctionnement de plusieurs services, chaque service ayant ses particularités. Le cabinet fournit alors un document XML contenant ces bilans. Ce document est ensuite traité par un logiciel qui établit un résultat personnalisé pour chaque service et propose également aux utilisateurs des fonctions de recherche et d'analyse.

Le document XML : orienté document ou données ?

Lorsque les données sont élaborées par des êtres humains, on dit que les fichiers XML produits sont orientés document. Lorsque les données sont construites automatiquement par des programmes, on dit que les fichiers XML sont orientés données. Un fichier XML orienté document peut être, par exemple, un livre, un article, un message… Un fichier XML orienté donnée est, par exemple, un sous-ensemble d'une base de données.

Il faut noter que l'élaboration des fichiers XML nécessite des moyens de contrôle et d'édition plus ou moins sophistiqués. On n'utilisera pas pour fabriquer un ouvrage en XML un éditeur trop rudimentaire (comme le bloc-notes sous l'environnement Windows). L'édition des documents XML sera abordée dans ce chapitre à la section L'édition de document XML.

La circulation XML : notion de bus

Les données informatiques circulent aussi bien en interne, dans l'entreprise, que vers l'extérieur, auprès de services et de partenaires externes. L'étendue de cette circulation rend le format de données d'autant plus important que chaque acteur peut disposer de plates-formes d'exploitation différentes. Le formalisme XML neutralise les différences par un consensus de stockage, la plupart des langages de programmation étant à même de traiter tout type de document XML. Les caractères Unicode constituent également un moyen de garantir la neutralité des données transportées.

Structure et validation d'un document XML

On associe à un document XML un schéma, qui peut être vu comme le schéma d'une base de données relationnelle. La validation d'un document XML garantit que la structure de données utilisée respecte ce schéma. On peut faire l'analogie avec le respect des règles d'orthographe et de grammaire d'une langue. Les documents XML qui circulent doivent ainsi être en accord avec ce schéma pour être acceptés par la plate-forme. Dans le cas contraire ils sont rejetés et doivent être refaits.

Lorsque les flux d'échanges sont denses, la validation peut présenter pour inconvénient de consommer des ressources. Il est difficile de raisonner pour tous les cas, mais la validation peut être considérée comme incontournable à certaines étapes de préparation du cadre d'exploitation. Lorsque les flux sont considérés comme stables, il est alors possible de pratiquer une forme d'assouplissement des règles dans l'optique d'améliorer les performances.

Transformation et adaptation d'un document XML

Un document XML peut être transformé ; il n'est pas figé par un émetteur mais peut suivre, par analogie avec les ateliers de production, différentes étapes de modification. Le format XSLT (*eXtensible Stylesheet Language Transformation*) est un moyen pour adapter un document XML à un autre format XML. Ces processus de transformation sont cependant coûteux et doivent répondre à un besoin. Conduire des transformations en

cascade peut être davantage pénalisant que de modifier les logiciels qui génèrent les documents XML, tout dépend de la réactivité souhaitée. Avec XSLT, on peut parfaitement imaginer exécuter la nuit des programmes *batch* qui réalisent ces générations de documents, l'une des générations possibles étant dans un langage de présentation comme XHTML ou bien XSL-FO (avec indirectement PDF, RTF…).

Par exemple, une société dispose d'un ensemble de produits. Ces produits sont présentés à la fois sur leur site Internet, dans un catalogue, et dans un logiciel interne pour les salariés… Le formalisme XML peut tisser un lien entre ces différents médias, les données étant au cœur de l'activité, la présentation n'étant plus qu'un processus de transformation.

Circulation des documents XML et workflows

Les flux de données (workflows) existants vont être petit à petit remplacés par des workflows XML. Les fichiers XML vont circuler, s'enrichir au fur et à mesure de ces déplacements, être contrôlés, puis être présentés aux différents acteurs de l'activité (commerciaux, clients…).

Prenons l'exemple d'un parc de machines équipées d'automates donc on souhaiterait contrôler l'activité. Comme il n'est pas possible de passer derrière chaque machine pour vérifier les opérations effectuées, un programme de type agent recueille les informations et les envoie au format XML à une borne de supervision.

Les bases de données

Les bases de données étant incontournables dans les systèmes informatiques actuels, nous allons, dans les paragraphes suivants, donner quelques points de repère quant à leurs relations avec XML.

XML et les bases relationnelles

Puisqu'il structure des données selon un schéma fixé, le formalisme XML peut-il remplacer les bases de données relationnelles telles que nous les connaissons ? La réponse est clairement non et c'est même le danger d'une mauvaise utilisation du formalisme XML. Un document XML est un fichier texte ; il n'est optimisé ni en espace ni pour les manipulations que l'on peut opérer sur ce type de fichiers. Un document XML pourrait être davantage perçu comme une partie d'un système d'information, car il résout un problème de circulation de l'information à un moment donné. Il n'y a pas de raison que les bases de données relationnelles ne soient pas gérées à l'avenir comme aujourd'hui. Tout au plus, nous pourrons voir l'apparition de solutions complémentaires. Par exemple, le typage des champs d'une table devrait offrir un typage XML à l'image du blob. La recherche par SQL sera peut-être étendue pour ces types *via* la solution XQuery ; le standard SQL ISO travaille sur SQL/XML (*http://www.sqlx.org/*). Quelques solutions existent déjà ça et là avec SQL Server ou Oracle, par exemple, mais ces solutions n'offrent pas encore de fonctionnement vraiment homogènes.

Les bases « natives » XML

L'autre aspect des relations entre les bases de données et le formalisme XML est l'utilisation de base de données « native XML ». C'est une solution séduisante pour agglomérer des documents et pouvoir les manipuler plus facilement. Cela peut compenser une certaine faiblesse à retrouver dans les tables des bases de données relationnelles la correspondance hiérarchique des documents XML. Et puis, les documents XML étant déjà structurés, l'idée de déstructurer ces documents en vue d'une insertion dans une base semble quelque peu inefficace.

On considère qu'il existe deux formes de bases de données natives : celles gardant le texte du document XML tel quel et celles effectuant une conversion sous une forme objet (comme DOM, qui est une standardisation objet d'un document XML). Il est certain que la deuxième forme peut s'appuyer sur des bases objets voire relationnelles (tables pour les éléments DOM : éléments, textes, commentaires…). Vous trouverez à l'adresse *http://www.rpbourret.com/xml/XMLDatabaseProds.htm* quelques bases de données natives, avec deux formes d'implémentation, O*pen Source* ou propriétaire. Parmi les bases disponibles, citons Tamino (*http://www.softwareag.com/Corporate/products/tamino/default.asp*) en propriétaire, ou bien XIndice (*http://xml.apache.org/xindice/*) en Open Source. Je n'ai pas de recommandation particulière à donner. Chaque base a ses avantages et inconvénients, en termes d'API d'accès, de langage de requêtes, de performance dans les traitements, l'objectif étant d'en mesurer l'efficacité sur un échantillon représentatif.

L'édition d'un document XML

L'édition de document XML peut prendre diverses formes, notamment en fonction de sa finalité.

Cas des formats orientés document

Pour réaliser un ouvrage, un article… en XML il n'est pas conseillé d'utiliser un éditeur de texte quelconque. La réalisation de tels documents impose de se focaliser sur le contenu et non sur la syntaxe du format de document. Pour arriver à alléger la part de ce travail, il existe des outils qui proposent l'édition en WYSIWYG (*what you see is what you get*) : l'auteur n'a alors plus l'impression de réaliser un document XML mais simplement d'utiliser un éditeur graphique (comme Word ou OpenOffice.org). Ces outils utilisent souvent une feuille de styles CSS (*Cascading StyleSheets*) qui donne une représentation graphique à telles ou telles parties du document XML. C'est pourquoi, certains logiciels proposent une édition XML via un navigateur de type Mozilla Firefox ou Internet Explorer.

Parmi les éditeurs Open Source WYSIWYG, citons Bitflux (*http://bitfluxeditor.org/*), Xopus (*http://xopus.com/*), qui utilise Internet Explorer et masque totalement la syntaxe XML, Serna (*http://www.syntext.com/products/serna/index.htm*), qui effectue un rendu à la frappe par XSLT et un sous-ensemble de XSL-FO et XMLMind, qui s'appuie sur des feuilles de styles (*http://www.xmlmind.com/xmleditor/*).

Les technologies XML s'intègrent dans les offres bureautiques notamment avec OpenOffice et Office 2007. Ces suites fonctionnent avec des formats incompatibles, respectivement Open Document et Open XML. Le format Open Document (1.1 au moment de l'écriture), pour la suite OpenOffice, a été réalisé par l'organisation OASIS (*Organization for the Advancement of Structured Information Standards*) et est normalisé ISO (ISO/IEC 26300:2006). Le format Open XML de la suite Office 2007 a été ratifié par l'organisme international ECMA (ECMA 376), il est en cours de normalisation ISO. Ces deux formats sont créés à base d'archive ZIP contenant un ensemble de fichiers XML (style, police, description, données, relation…) et d'autres ressources binaires liées (images, audio..). Bien qu'ils soient incompatibles, il existe un traducteur imparfait s'appuyant sur des transformations XSLT que l'on peut trouver à l'adresse suivante : *http://odf-converter.sourceforge.net*. Le format Open Document s'appuie davantage sur des standards (RDF, SVG, MathML) que sur Open XML. On peut ainsi reprocher à ce dernier de s'inscrire dans la continuité par rapport aux formats Microsoft Office tout en sachant que cela représente aussi la réalité du marché. Les différents outils de la suite de Microsoft s'associent avec des schémas W3C. Ces schémas servent à agglomérer, modifier, importer et exporter des documents XML par exemple dans une feuille Excel ou une page Word. Des transformations XSLT pendant les opérations de lecture ou d'écriture sont également possibles ; elles donnent la possibilité de visualiser différemment le document sous différentes vues. À noter que Microsoft propose également le format XPS (*XML Paper Specification*) sous la forme d'un complément à télécharger pour la suite Office 2007. Ce dernier est un concurrent de PDF ou de Postscript mais en version XML. Un lecteur XPS est également disponible sur le site de Microsoft *(http://www.microsoft.com/whdc/xps/viewxps.mspx*). Il faut noter la présence avec Adobe du format XDP (*XML Data Package*) comme solution XML, probablement en remplacement progressif du format PDF, de la même façon que ce dernier a, peu à peu, éclipsé le format Postscript.

Cas des formats orientés données

Dans ce type de format, il n'y a pas de représentation facilement utilisable pour l'être humain, l'idéal étant de passer par une application qui masquera la localisation des données.

Édition avec un formulaire

Certaines solutions visent à analyser les schémas des fichiers XML pour générer un formulaire de saisie. Cela peut être intéressant lorsque ce formulaire est disponible *via* un navigateur.

Parmi les éditeurs proposant cette solution, citons EditLive! (*http://www.ephox.com/*) et Microsoft, avec InfoPath (*http://office.microsoft.com/en-us/infopath/default.aspx*).

Éditeurs plus généralistes

Les éditeurs généralistes sont une autre forme d'éditeurs qui s'adressent plutôt à des techniciens. Il existe de nombreux produits, qui offrent tous la validation et la transformation. Ils se démarquent par certaines facilités.

Le plus connu est l'éditeur XMLSpy (*http://www.altova.com/*) pour un prix minimum de 399 euros par licence. Ce logiciel a gagné en réputation avec son éditeur de schémas W3C WYSIWYG. Il dispose également d'une édition par grille. En dehors de cette facilité, l'interface est quelque peu vieillotte et offre peu d'assistants à la frappe.

Stylus Studio est un autre éditeur proche de XMLSpy (*http://www.stylusstudio.com/*) dont le prix minimum est d'environ 300 euros par licence. Cette application a une certaine réputation pour son éditeur XSLT semi-WYSIWYG. En dehors de cette facilité, l'édition manque souvent d'assistants et l'interface manque d'ergonomie.

EditiX (*http://www.editix.com/*) est un éditeur dont le prix minimum est de 70 euros par licence, qui offre une vérification syntaxique à la frappe. Une arborescence est à tout moment synchronisée avec le texte, ce qui facilite grandement la navigation. L'éditeur n'offre pas de mode WYSIWIG mais contient de nombreux assistants en fonction des documents édités. C'est cet éditeur que nous avons utilisé pour réaliser les travaux pratiques de cet ouvrage. Vous disposez d'une version d'évaluation de 30 jours (*http://www.editix.com/download.html*).

XMLCooktop (*http://www.xmlcooktop.com/*) est l'éditeur gratuit le plus connu. Attention, car s'il peut suffire pour des tâches XML simples, ses limitations et l'absence de maintenance (abandon du développement annoncé par l'auteur) rendent son utilisation délicate dans un contexte professionnel.

XMLNotepad 2007 (*http://msdn.microsoft.com/xml*) est un éditeur gratuit mis à disposition sur la plate-forme Windows. Il semble intéressant pour des documents de grandes tailles mais offre très peu d'aide à la saisie.

Outils pour manipuler les documents XML

Les parseurs XML

Un *parseur* a pour rôle d'analyser le document XML et de servir de lien avec une application de traitement. Il existe des parseurs non validants qui n'offrent qu'une vérification syntaxique et des parseurs validants qui offrent également le support des DTD/schéma W3C. Sur ces deux catégories de parseurs se greffent principalement deux catégories de services : un service événementiel, qui ne vise pas à représenter un document XML dans son intégralité, de type SAX (*Simple API for XML*), par exemple, et un service objet, qui permet de représenter un document XML sous une forme objet, de type DOM (*Document Object Model*), par exemple. Dans le premier cas, la représentation du document n'est que partielle, alors que dans le second cas, elle est complète. Ces deux méthodes ont leurs avantages et inconvénients. Citons seulement la consommation mémoire et la facilité des traitements (requêtage…). Ces concepts seront étendus dans le chapitre dédié à la programmation.

Microsoft XML Core Services (MSXML : *http://msdn.microsoft.com*) est une API composée d'un parseur validant, compatible SAX et DOM, et d'un moteur de transformation 1.0.

Xerces est disponible pour Java, C++ et Perl. C'est un logiciel Open Source réalisé par le groupe apache (*http://xerces.apache.org/*). Il s'agit probablement du parseur le plus abouti du marché, quelle que soit la plate-forme, en terme de respect du standard et de l'API (SAX, DOM). Ses performances sont aussi remarquables.

Un certain nombre de plates-formes, comme PHP et Java, disposent d'un parseur en standard.

Expat est un parseur réalisé en C (*http://expat.sourceforge.net/*), utilisé par le projet Mozilla. Il dispose d'extensions pour SAX et DOM. Un ensemble de tests (*benchmark*) le présente comme beaucoup plus rapide que les autres parseurs (résultats disponibles à l'adresse *http://www.xml.com/pub/a/Benchmark/article.html?page=3*).

Piccolo est un parseur non validant réalisé en Java (*http://piccolo.sourceforge.net/*). Les benchmarks disponibles, qui le présentent comme performant (*http://piccolo.sourceforge.net/bench.html*), peuvent être trompeurs car ils prennent en compte d'anciennes versions des autres parseurs ; par exemple, les dernières versions de Xerces donnent de meilleures performances.

Transformation d'un document XML

La transformation XSLT d'un document XML fonctionne en complément d'un parseur. Il s'agit d'une API qui réalise le passage d'un document XML vers un document texte (souvent au format XML lui aussi). La plupart des moteurs de transformation ne gèrent que la version XSLT 1.0.

Le toolkit MSXML de Microsoft (*http://msdn.microsoft.com/*) supporte la version 1.0.

Le groupe Apache gère le projet Xalan (*http://xalan.apache.org/*) pour Java et C++ avec support de la version 1.0.

Saxon est un projet Open Source avec également une licence commerciale. Il fonctionne pour Java et .NET et gère les versions 1.0 et 2.0.

Sablotron est une implémentation en C++ de la version 1.0 (*http://www.gingerall.org/sablotron.html*). Il peut être employé sous forme d'extension en PHP, Perl, Pascal…

Le format XSL-FO

XSL-FO (*Extensible Stylesheet Language Formatting Objects*) est un langage de présentation pour différents formats (PDF, RTF...).

Il y a peu d'outils à l'heure actuelle capables de réaliser les transformations XSL-FO.

Une première solution propriétaire est Ecrion (*http://www.ecrion.com/*). Elle gère en sortie les formats PDF et PostScript.

Une autre solution propriétaire est XEP de RenderX (*http://www.renderx.com/*). Elle gère en sortie les formats PDF et PostScript.

La seule solution Open Source est probablement FOP (*Formatting Objects Processor*) du groupe Apache (*http://xmlgraphics.apache.org/fop/*). Elle gère en sortie les formats PDF, PostScript et RTF.

Le format SVG

SVG (*Scalable Vector Graphics*) est un langage de description des dessins en 2D. Il existe quelques plug-ins pour les navigateurs, dont une intégration native avec Firefox 2.0, le plus connu étant SVG Viewer de adobe (*http://www.adobe.com/svg/viewer/install/main.html*). Attention cependant, l'éditeur annonçant la fin du support pour 2008.

On retiendra comme implémentation Open Source le projet Batik pour Java (*http://xmlgraphics.apache.org/batikl*) du groupe Apache.

2

Structure des documents XML

Ce chapitre vous apprend à structurer un document XML. La compréhension de ses composants est indispensable pour aborder les grands principes de XML.

Structure d'un document XML

Commençons par prendre un exemple simple de document XML :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Date de création : 30/09/07 -->
<cours titre="XML">
    <intervenant nom="alexandre brillant">
    </intervenant>
    <plan>
        Introduction
        XML et la composition de documents
    </plan>
</cours>
```

On peut d'ores et déjà analyser qu'un document XML est un document texte lisible. Sans comprendre nécessairement l'intégralité de la syntaxe, on en déduit qu'il s'agit de la description d'un cours dont l'intervenant n'est autre que l'auteur de cet ouvrage. Le plan du cours ressort également du document.

Nous allons maintenant décortiquer la syntaxe et en comprendre les tenants et les aboutissants.

L'en-tête : le prologue

Il s'agit de la première ligne d'un document XML servant à donner les caractéristiques globales du document, c'est-à-dire :

- La version XML, soit 1.0 ou 1.1, sachant que la très grande majorité des documents sont en version 1.0 et que la version 1.1 est assez décriée (recommandation du W3C en version 2 du 4 février 2004, qui note la résolution d'une incompatibilité avec les *mainframes* IBM).
- Le jeu de caractères employé (*encoding*). Pour fonctionner, le parseur va avoir besoin de distinguer le rôle de chaque caractère, certains étant réservés à la syntaxe et d'autres représentant des données. Pour définir un jeu de caractères, XML s'appuie sur des standards ISO et Unicode (voir *http://www.unicode.org/*). Notre standard Europe de l'ouest (ou iso-latin) est qualifié par ISO-8859-1. Lorsque l'encodage n'est pas précisé, c'est le standard UTF-8 qui est employé (avantage d'une compatibilité ANSI). Il existe beaucoup d'autres standards, citons ISO-2022-JP, Shift_JIS, EUC-JP… UTF-16 a la particularité de nécessiter l'encodage de chaque caractère avec 2 octets (un même fichier sera donc deux fois plus lourd encodé en UTF-16 qu'en UTF-8).
- Le champ `standalone` désigne l'indépendance du document, au sens où il n'existe aucun élément externe qui puisse altérer la forme finale du document XML fourni à l'application par le parseur (références d'entités, valeurs par défaut…). Ce champ prend les valeurs `yes` ou `no`. Dans la pratique, au-delà de l'aspect informatif, il n'a pas grand intérêt et vous pouvez l'ignorer.

Si nous reprenons notre exemple précédent, nous avons donc comme prologue :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

À noter que ce prologue est encapsulé par `<?` et `?>` et qu'il n'existe pas d'espaces (de blancs) entre le début du document et cet élément. Autre remarque qui peut surprendre : le prologue n'est pas obligatoire et, dans ce cas, le parseur utilise un comportement par défaut (version 1.0 et encoding UTF-8).

Les instructions de traitement

Les instructions de traitement (*processing instruction* ou PI) n'ont pas de rôle lié aux données ou à la structuration de votre document. Elles servent à donner à l'application qui utilise le document XML des informations. Ces dernières sont totalement libres et dépendent avant tout du concepteur de l'application de traitement. On les positionne à n'importe quel endroit du document (après le prologue, bien entendu). Un cas typique est l'utilisation avec les navigateurs Mozilla Firefox ou Internet Explorer pour effectuer la transformation d'un document XML en document XHTML affichable avec l'instruction :

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

L'emploi de cette instruction et le rôle du document `affichage.xsl` seront éclairés dans la partie consacrée aux feuilles de styles.

> **Remarque**
>
> Attention à ne pas confondre ces instructions de traitement avec le prologue qui, s'ils sont de syntaxe similaire, n'ont pas le même rôle.

Enfin, ces instructions de traitement ont également pour fonction de pallier des manques dans la spécification XML, par exemple pour effectuer un lien vers un langage de validation non standard (vérifiant donc la conformité du document par rapport à une grammaire) de type Relax NG (*http://www.relaxng.org/*).

Les commentaires

Il y a peu de chose à dire sur les commentaires. Ce sont les mêmes qu'en HTML (ceci est dû au lien de parenté avec SGML). Ils se positionnent n'importe où après le prologue et peuvent figurer sur plusieurs lignes.

Notre exemple contient le commentaire suivant :

```
<!-- Date de création : 30/09/07 -->
```

Point important : les caractères `--` sont interdits comme commentaires pour une raison que l'on comprendra aisément (ambiguïté d'analyse pour le parseur).

La déclaration du type de document

Cette déclaration optionnelle sert à attacher une grammaire de type DTD (*Document Type Definition*) à votre document XML. Elle est introduite avant la première balise (racine) de votre document sous cette forme :

```
<!DOCTYPE racine SYSTEM "URI vers la DTD">
```

`racine` est le premier élément (la première balise). L'URI peut être absolue ou relative au document. Il est généralement préférable soit d'utiliser une URI relative, pour pouvoir déplacer le document XML et sa grammaire sans difficulté, ou bien d'exploiter une URL disponible depuis n'importe quel endroit (sur Internet/Intranet, par exemple).

Exemple :

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

Dans cet exemple, la DTD `cours.dtd` est localisée relativement à notre document XML.

Le mot-clé `SYSTEM` est important et indique qu'il s'agit d'une DTD qui vous est propre. L'alternative est le mot-clé `PUBLIC`.

Exemple :

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0 Strict//EN" >
```

Ici, l'URI est remplacée par un identifiant public composé de :

- - : indique qu'il s'agit d'un format non compatible ISO (sinon on utilise +).
- IETF : organisme gérant le format.
- DTD : indique la nature du document.
- HTML 2.0 Strict : description du document.
- EN : un code langue (ISO 639).

On peut également préciser, après l'identifiant public, une URI. C'est utile si le parseur ne peut pas traduire l'identifiant public en URI exploitable.

La déclaration de type de document peut également héberger un bloc d'instructions propre aux DTD (on parle alors de DTD interne). Sans rentrer dans les détails, puisque cela sera repris dans l'analyse des DTD dans un prochain chapitre, voici un exemple de DTD interne qui sert à valider notre document XML :

```
<!DOCTYPE cours [
<!ELEMENT cours ( intervenant, plan )>
<!ELEMENT intervenant EMPTY>
<!ELEMENT plan (#PCDATA)>
<!ATTLIST cours titre CDATA #REQUIRED>
<!ATTLIST intervenant nom CDATA #REQUIRED>
]>
```

Les nœuds élément

Les éléments gèrent la structuration des données d'un document XML, un peu à la manière des répertoires qui servent à l'organisation des fichiers. On peut les qualifier de métadonnés, au sens où ils ne font pas partie réellement des données mais servent à en désigner la nature. À la place du terme élément, on peut utiliser les termes balise, *tag* ou encore nœud.

Pour se familiariser rapidement, reprenons l'exemple précédent. À l'intérieur, nous trouvons les éléments `cours`, `intervenant` et `plan`.

Pour décrire ce que contiennent les éléments, on parle de modèle de contenu. On trouve :

- Rien : il n'y pas de contenu, l'élément est vide.
- Du texte : nous détaillerons par la suite cette notion.
- Un ou plusieurs éléments : on peut les qualifier d'éléments fils, l'élément les contenant étant appelé un élément parent (à ne pas confondre avec un élément ancêtre,

qui indique qu'il existe une relation de conteneur à contenu et qui est donc plus large).

- Un mélange de textes et d'éléments : c'est une forme plus rare qui peut révéler une erreur de structuration. Elle reste cependant utile, lorsque l'on souhaite « décorer » un texte quelconque (cas du paragraphe en HTML avec des zones en gras, italique…).

Reprenons notre exemple XML et complétons-le un peu :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<cours>
   <intervenant>
   Phileas
   </intervenant>
   <separateur/>
   <chapitre>
   Formation XML
       <para>Un paragraphe</para>
       <para>Autre paragraphe</para>
   </chapitre>
</cours>
```

- `cours` : élément racine contenant trois éléments fils : `intervenant`, `separateur` et `chapitre` ;
- `intervenant` : élément contenant du texte ;
- `separateur` : élément sans contenu ;
- `chapitre` : élément contenant du texte et des éléments fils `para` ;
- `para` : élément contenant du texte.

Remarque

Nous effectuons une mise en page minimale à l'aide de tabulations (on parle d'indentation) afin de faire ressortir la structure du document, comme les relations parent/enfant.

Si maintenant nous nous penchons sur la syntaxe, nous avons donc :

- `<element>` : balise ouvrante.
- `</element>` : balise fermante.
- `<element/>` : balise ouverte et fermée que l'on nomme balise autofermée. C'est l'équivalent de `<element></element>`. Elle désigne donc un élément vide.

Remarque

Cette dernière forme est propre à XML et n'apparaît pas dans HTML.

Exercice 1

Création d'un livre en XML

On souhaite écrire un livre en utilisant le formalisme XML. Le livre est structuré en sections (au moins 2), en chapitres (au moins 2) et en paragraphes (au moins 2).

Le livre doit contenir la liste des auteurs (avec nom et prénom).

Tous les éléments doivent posséder un titre, sauf le paragraphe qui contient du texte.

Proposez une structuration XML de ce document (avec 2 auteurs, 2 sections, 2 chapitres par section et 2 paragraphes par chapitre).

Vérifiez, à l'aide de l'éditeur, que votre document est bien formé.

Attention : ne pas utiliser d'attributs ; l'encodage utilisé est `ISO-8859-1`

Votre document sera nommé `livre1.xml`.

Les attributs d'un élément

Un attribut est un couple (clé, valeur) associé à la définition d'un élément. Il est localisé dans la balise ouvrante de l'élément. Un élément peut donc avoir de 0 à n attributs uniques. L'attribut est complémentaire de l'élément de par son rôle au sens où il ajoute une information à l'élément ou bien encore le complète dans sa définition.

Exemple :

```
<auteur nom="brillant" prenom="alexandre">…</auteur>
<contact email='a@a.fr'/>
```

`nom` et `prenom` sont des attributs de l'élément `auteur` alors que `email` est un attribut de l'élément `contact`.

On sépare les attributs par au moins un espace (blanc simple, tabulation, retour à la ligne). Les valeurs d'attributs peuvent figurer sur plusieurs lignes. On utilise soit les guillemets, soit les apostrophes pour encapsuler les valeurs.

Voici un exemple de document XML avec des attributs :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<cours>
   <intervenant nom="fog" prenom="phileas"/>
   <introduction/>
   <chapitre numero="1">
       Formation XML
       <paragraphe>Détails du format</paragraphe>
   </chapitre>
</cours>
```

Choix entre éléments et attributs

L'attribut peut sembler superflu. En effet, ce qui s'écrit avec des attributs peut également l'être en s'appuyant uniquement sur des éléments.

Exemple :

Cas avec attributs :

```
<personne nom="brillant" prenom="alexandre"/>
```

Cas sans attribut :

```
<personne>
   <nom>
       brillant
   </nom>
   <prenom>
       alexandre
   </prenom>
</personne>
```

Cependant, l'inverse n'est pas vrai car un attribut ne peut pas être répété dans un élément (mais il peut l'être au travers d'éléments différents).

Exemple :

Cas avec éléments :

```
<carnet>
   <personne>…
   </personne>
   <personne>…
   </personne>
</carnet>
```

S'il fallait supprimer les éléments `personne` au profit d'attributs, il faudrait utiliser une convention de nommage complexe des attributs (avec une numérotation pour chaque personne…) ce qui ferait perdre tout intérêt à XML qui sert justement à structurer des données.

On peut définir cependant quelques règles simples pour déterminer s'il est préférable d'utiliser un attribut ou un élément. Lorsqu'une valeur est de taille modeste, a peu de chance d'évoluer vers une structure plus complexe, et n'est pas répétée, alors l'attribut peut tout à fait convenir. Dans tous les autres cas, l'élément reste incontournable. Si vous avez un doute entre un attribut et un élément, alors choisissez toujours l'élément qui est le plus ouvert et garantira le plus facilement une évolution.

Exercice 2

Utilisation des attributs

Conception de livre2.xml à partir de livre1.xml

On souhaite compléter la structure du document XML de l'exercice précédent par les attributs `nom` et `prenom` pour les auteurs et `titre` pour le livre, les sections et les chapitres.

Analysez la structure du nouveau document. Y a-t-il des simplifications possibles ?

Vérifiez, à l'aide de l'éditeur, que votre document est bien formé.

Les nœuds textes

Dans un document XML, ce qui est appelé donnée est le texte qui est associé à l'attribut, c'est-à-dire sa valeur, ou à l'élément, c'est-à-dire son contenu. Les données constituent le cœur du document, et tout le reste, le formalisme, ne sert qu'à séparer et classer ces données. Celles-ci sont dites terminales dans l'arborescence XML, dans la mesure où il ne peut y avoir de structuration supplémentaire. Dans le vocabulaire propre à DOM (*Document Object Model*), que nous aborderons dans un autre chapitre, la donnée apparaît comme un nœud texte. Elle est bien entendu liée au prologue puisque l'encodage sert à désigner le jeu de caractères utilisé dans votre document.

Dans le premier exemple XML que nous avons donné, le texte `XML` est une donnée liée à l'attribut `titre` de l'élément `cours`. `Introduction…` est une donnée liée à l'élément `plan`.

Comme certains caractères sont réservés à la syntaxe XML, il faut être vigilant lors de l'écriture des données.

Exemple :

```
<calcul>
   if ( a<b et b>c) ...
</calcul>
```

Dans cet exemple, on voit bien que `<b et b>` n'est pas une balise mais fait partie des données liées à l'élément `calcul`. Ce que nous comprenons facilement n'est pas sans ambiguïté pour un parseur qui identifiera une balise de syntaxe incorrecte.

Pour résoudre ce problème, nous disposons d'entités prédéfinies. L'entité en elle-même peut être perçue comme un raccourci vers une autre valeur, ce qui a l'avantage de faciliter la maintenance (changement de valeur avec répercussions) et de masquer les conflits syntaxiques.

Voici la liste des entités prédéfinies :

- `<` équivalent de < (*less than*) ;
- `>` équivalent de > (*greater than*) ;
- `&` équivalent de & (*ampersand*) ;

- `&quot;` équivalent de " (*quote*) ;
- `&apos;` équivalent de ' (*apostrophe*).

L'exemple précédent peut donc être correctement réécrit :

```
If (a&lt;b et b&gt;c)
```

Bien entendu, les entités prédéfinies ne servent qu'à lever une ambiguïté syntaxique pour le parseur.

Ces entités peuvent être utilisées dans un élément, pour du texte, ou dans un attribut, pour sa valeur.

Les entités prédéfinies présentes en trop grand nombre dans un même bloc peuvent alourdir inutilement le document. Dans le cas du contenu textuel d'un élément (et uniquement dans ce cas), nous disposons des sections CDATA (*Character Data*). Cette section doit être considérée comme un bloc de texte dont les caractères seront pris tel quel par le parseur jusqu'à la séquence de fin `]]>`.

Exemple :

```
<![CDATA[
<element>C'est un document XML
</element>
]]>
```

Dans cet exemple, `<element>` n'est pas considéré comme une balise de structuration, mais comme du texte.

Exercice 3

Utilisation des entités prédéfinies

On se propose de créer un nouveau document `livre2bis.xml` reprenant l'exercice précédent (`livre2.xml`). Placez dans 2 paragraphes un bloc de texte contenant l'extrait suivant :

```
<element id="10">&gt;</element>
```

Pour le premier paragraphe, employez les entités prédéfinies.

Pour le deuxième paragraphe, employez une section CDATA.

Les entités du document

Nous avons déjà vu une première forme d'entité à travers les entités prédéfinies. Si vous concevez une DTD (*Document Type Definition*), sachez que vous pourrez à loisir créer votre propre entité et y faire référence dans vos documents. Nous aborderons la syntaxe des DTD dans un prochain chapitre, mais en attendant, voici un exemple d'utilisation :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE cours [
<!ENTITY auteur "Alexandre Brillant">
```

```
<!ELEMENT cours (#PCDATA)>
]>
<cours>
   Cours réalisé par &auteur;
</cours>
```

L'entité `auteur` est liée à un nom et un prénom.

La valeur de l'entité peut être interne ou externe (placée dans un autre fichier). Attention à ne pas confondre les entités que l'on trouvera dans un document HTML (comme ` `) avec celles que vous aurez à votre disposition. Dans les deux cas il s'agit bien d'une DTD qui en délimite l'utilisation. Si votre DTD ne comporte pas d'entités identiques à celles possibles dans un document HTML alors vous ne pourrez pas les employer.

Il existe également une forme d'entités, qui ne sert qu'à la DTD, appelée entités paramétriques. Nous les aborderons dans un prochain chapitre.

Enfin, il reste l'entité caractère. Cette entité va nous être utile pour employer un caractère issu de la table ISO/IEC 10646 que l'on retrouve également avec HTML (proche de UTF-16). On spécifie la valeur du caractère soit en décimal soit en hexadécimal.

Par exemple, pour représenter un retour à la ligne sous Linux/Unix (il faut en plus le caractère 13 pour la plate-forme Windows), on utilise :

```
Forme décimale :
&#10;
Forme hexadécimale :
&#xA;
```

Quelques règles de syntaxe

Ces règles de syntaxe sont à respecter impérativement pour qu'un document XML soit bien formé.

- Le nom d'un élément ne peut commencer par un chiffre.
- Si le nom d'un élément est composé d'un seul caractère il doit être dans la plage [a-zA-Z] (c'est-à-dire une lettre minuscule ou majuscule sans accent) ou _ ou :.
- Avec au moins 2 caractères, le nom d'un élément peut contenir _, -, . et : plus les caractères alphanumériques (attention, le caractère : est réservé à un usage avec les espaces de nom que nous aborderons par la suite).
- Tous les éléments ouverts doivent être fermés.
- Un élément parent est toujours fermé après la fermeture des éléments fils. Voici un contre-exemple où l'élément fils `b` est incorrectement fermé après la fermeture de son élément parent `a` : `<a><b></a></b>`.

Pour connaître plus précisément les caractères autorisés, vous pouvez vous reporter à la grammaire XML du W3C disponible à l'adresse *http://www.w3.org/TR/2006/REC-xml-20060816/#sec-well-formed*.

Quelques conventions de nommage

Voici quelques conventions souvent employées dans les documents XML :

- Employer des minuscules pour les attributs et les éléments.
- Éviter les accents dans les noms d'attributs et d'éléments pour des raisons de compatibilité avec les outils du marché qui proviennent souvent d'un univers anglo-saxon.
- Préférer les guillemets délimitant les valeurs d'attribut.
- Séparer les noms composés de plusieurs mots par les caractères -, _, . ou une majuscule. Essayer d'être homogène dans votre document en gardant la même convention.

Exemples :

```
<value-of/>
<valEntier></valEntier>
```

Quelques exemples XML

Voici quelques exemples simples de documents XML. Certaines formes de structuration d'usage courant ont été normalisées par différents organismes comme le W3C ou le consortium OASIS (*http://www.oasis-open.org*). Vous pouvez à titre d'exercice essayer de distinguer chaque composante de ces documents.

Le format MathML

MathML est une recommandation du W3C (*http://www.w3.org/Math/*) servant à d'écrire des expressions mathématiques.

```
<?xml version="1.0"?>
<math>
 <mrow>
  <msup> <mi>x</mi><mn>2</mn> </msup>
  <mo>+</mo>
  <mrow>
    <mn>4</mn><mo>&InvisibleTimes;</mo><mi>x</mi>
  </mrow>
  <mo>+</mo>
  <mn>4</mn>
 </mrow>
</math>
```

Très brièvement, `mo` désigne un opérateur, `mrow` une expression, `mi` et `mn` respectivement des opérandes variable (par exemple `x`) et nombre.

Ici nous avons décrit l'expression : `x^2+4x+4`.

Le format VoiceXML

VoiceXML est un standard pour décrire des choix (processus de questions/réponses) liés à des répondeurs vocaux (*http://www.voicexml.org/*).

```
<vxml version="2.0">
<menu>
  <prompt>Faire un choix :<enumerate/></prompt>
  <choice next="http://www.meteo.exemple/meteo.vxml">Meteo
  </choice>
  <choice next="http://www.infos.exemple/infos.vxml">Info
  </choice>
  <noinput>Merci de faire un choix <enumerate/></noinput>
</menu>
</vxml>
```

Très brièvement, `menu` représente une liste de choix, `prompt` est une question, `choice` une réponse possible et `noinput` la phrase produite si aucun choix n'est réalisé ; le menu est rappelé avec `enumerate`.

Les espaces de noms

Les espaces de noms sont un concept très commun en informatique. Nous les retrouvons dans de nombreux langages de programmation afin de prévenir d'éventuels conflits. Par exemple, dans le langage de programmation Java, les *packages* servent à délimiter la portée d'une classe. En choisissant un package, le développeur lève tout conflit potentiel sur le nom, car la classe sera par la suite utilisée directement ou indirectement via son package et son nom.

Application des espaces de noms dans un document XML

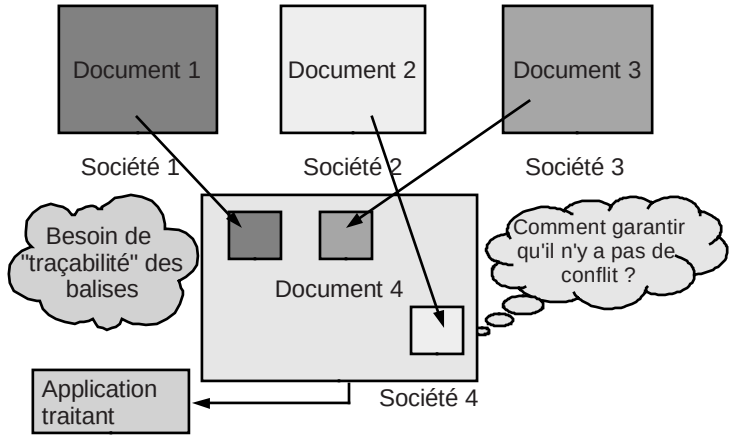
Lorsque nous créons un document XML, la structure du document est déterminée à partir des données que nous allons avoir à gérer. Mais ce que nous ne pouvons pas prévoir, ce sont les données d'origines diverses qui peuvent être en contradiction avec nos choix de structure initiaux. Il n'est pas non plus impossible que nous ayons à recouper plusieurs documents XML afin de produire un document de synthèse. Toutes ces opérations peuvent introduire autour de nos balises et attributs des conflits potentiels, car une balise ou un attribut peut avoir du sens dans un contexte mais pas dans un autre. C'est d'autant plus vrai que, ne l'oublions pas, un document XML est traité par une application qui ne doit pas rencontrer d'ambiguïté. Comment garantir, quand nous choisissons une balise ou un attribut, que personne n'a eu la même idée dans un contexte d'application différent ? Par exemple, le choix de l'élément `titre` peut représenter le titre d'un livre, mais pourquoi pas le titre de noblesse d'une personne…

Pour délimiter la portée d'une balise, d'un attribut ou d'une valeur d'attribut, nous disposons d'espaces de noms (*namespace*). Par analogie, cette notion est reprise dans la plupart des langages de programmation récents, comme les packages en java ou les

namespaces dans l'environnement .net. La notion d'espace de noms peut être perçue comme un groupe d'appartenance ou une famille. L'utilisation des espaces de noms garantit une forme de traçabilité de la balise et évite les ambiguïtés d'usage.

Les espaces de noms sont très souvent employés dans les spécifications W3C car ces documents peuvent être mélangés à d'autres et entraîner des conflits. On en trouve, par exemple, dans les feuilles de styles (XSLT) et dans les schémas W3C.

Figure 2-1
Croisement de documents



Dans la figure 2-1, nous avons 3 documents provenant de sociétés différentes. Ces trois documents sont intégrés dans un document utilisé par la société 4. Il est impératif que ces mélanges se fassent au mieux et sans conflit. Il peut être également intéressant, pour l'application de traitement, de savoir différencier ce qui vient de telle ou telle société.

Pour que les espaces de noms aient un sens, il faut pour chacun d'eux un identifiant unique. Dans le cas contraire, les espaces de noms pourraient eux-mêmes être en conflit. Cet identifiant unique peut être simplement l'URL, puisqu'il ne peut y avoir qu'un propriétaire pour une URL donnée (ceci est lié au nom de domaine présent dans l'URL et à son obtention auprès d'un organisme comme l'AFNIC en France). C'est un peu similaire à un bureau d'enregistrement.

> **Attention**
> L'URL ne signifie pas qu'il doit y avoir un document sur votre serveur HTTP. Ce n'est qu'un identifiant et n'importe quelle chaîne de caractères pourrait en réalité être employée.

Vocabulaire : qualification des éléments

Un élément qui est connu dans un espace de noms est dit qualifié ; dans le cas contraire, il est dit non qualifié. Lorsqu'on ignore l'espace de noms d'un élément, on dit qu'on s'intéresse à sa forme locale. Ce vocabulaire est à connaître lorsqu'on travaille avec un parseur dans les techniques dites SAX ou DOM.

Utilisation des espaces de noms dans un document XML

Il existe plusieurs méthodes pour intégrer des espaces de noms. Nous allons aborder les règles implicites et explicites sachant que cette dernière solution est davantage employée.

L'espace de noms par défaut

Un premier usage consiste à utiliser simplement l'espace de noms par défaut. Ce dernier est précisé par un pseudo-attribut `xmlns` (retenir que `ns` est pour *namespace)*. La valeur associée sera une URL garantissant l'unicité de l'espace de noms. L'espace de noms par défaut s'applique à l'élément où se situe sa déclaration et à tout son contenu.

Exemple :

```
<chapitre xmlns="http://www.masociete.com">
   <paragraphe>
   ...
   </paragraphe>
</chapitre>
```

Ici l'élément `chapitre` est dans l'espace de noms `http://www.masociete.com`. C'est également le cas de l'élément `paragraphe`, puisqu'il est dans l'élément `chapitre`.

Nous pouvons changer l'espace de noms par défaut même dans les éléments enfants : dans ce cas, une règle de priorité est appliquée. Attention, les espaces de noms ne sont pas imbriqués ; on ne peut appliquer qu'un seul espace de noms à la fois.

Exemple :

```
<chapitre xmlns="http://www.masociete.com">
   <paragraphe xmlns="http://www.autresociete.com">
   ...
   </paragraphe>
</chapitre>
```

L'élément `paragraphe` n'appartient pas à l'espace de noms `http://www.masociete.com` mais uniquement à l'espace de noms `http://www.autresociete.com`.

> **Attention**
> Un espace de noms par défaut ne concerne que les éléments. Les attributs et les textes n'y appartiennent pas. Le texte d'un élément n'est jamais dans un espace de noms puisqu'il représente la donnée.

L'espace de noms explicite

L'espace de noms par défaut présente l'inconvénient d'être peu contrôlable sur un document de taille importante. En effet, tout ajout ou modification d'un tel espace va se répercuter sur la totalité du contenu.

Pour disposer de davantage de souplesse dans ces espaces et pouvoir également les appliquer aux attributs et valeurs d'attributs, la syntaxe introduit la notion de préfixe.

Un préfixe est une sorte de raccourci vers l'URL de l'espace de noms. On peut faire l'analogie avec une forme de macro ou de constante. Autre analogie, dans le langage SQL : il arrive que des champs de même nom se trouvent dans des tables différentes et qu'une opération de jointure entre ces tables oblige à distinguer chaque champ en le préfixant par un alias lié à chaque table.

On déclare un préfixe comme un pseudo-attribut commençant par `xmlns:prefixe`. Une fois déclaré, il est employable uniquement dans l'élément le déclarant et dans son contenu. L'emploi consiste à ajouter en tête de l'élément, de l'attribut ou d'une valeur d'attribut, le préfixe suivi de `:`.

Exemple :

```
<p:resultat xmlns:p="http://www.masociete.com">
</p:resultat>
```

L'élément `resultat` est dans l'espace de noms `http://www.masociete.com` grâce au préfixe `p`.

Attention

Lorsqu'un élément est préfixé, son contenu doit l'être aussi si l'on souhaite que l'espace de noms s'applique également.

La notion de préfixe n'a de sens que par rapport à l'URL associée. Dans la pratique, l'application ne voit pas ce préfixe mais uniquement l'URL associée à telle ou telle partie du document. C'est comme si un élément était un couple (nom de l'élément, espace de noms), de façon analogue à un attribut et sa valeur.

Exemple :

```
Document 1 :
<p:res xmlns:p="http://www.masociete.com">
</p:res>
Document 2 :
<zz:res xmlns:zz="http://www.masociete.com">
</zz:res>
```

Les documents 1 et 2 sont strictement identiques malgré un préfixe différent ; `res` étant dans tous les cas un élément appartenant à l'espace de noms `http://www.masociete.com`.

On peut déclarer et utiliser plusieurs espaces de noms grâce aux préfixes.

Exemple :

```
<p:res xmlns:p="http://www.masociete.com" xmlns:p2="http://www.autresociete.com">
   <p2:res>
   </p2:res>
</p:res>
```

Le premier élément `res` est dans l'espace de noms `http://www.masociete.com` alors que l'élément `res` à l'intérieur est dans l'espace de noms `http://www.autresociete.com`.

Attention

On ne peut pas utiliser plusieurs préfixes en même temps sur un élément, attribut ou valeur d'attribut (exemple à ne pas faire `p:p2:res`).

La suppression d'un espace de noms

Aucun espace de noms n'est utilisé lorsqu'il n'y a pas d'espace de noms par défaut ni de préfixe.

Exemple :

```
<p:element xmlns:p="http://www.masociete.com">
   <autreelement/>
</p:element>
```

L'élément `element` est dans l'espace de noms `http://www.masociete.com` alors que l'élément `autreelement`, qui n'est pas préfixé, n'a pas d'espace de noms.

Pour supprimer l'action d'un espace de noms il suffit d'utiliser la valeur vide `""`, ce qui revient à ne pas avoir d'espace de noms.

Exemple :

```
<element xmlns="http://www.masociete.com">
   <autreelement xmlns="">
       .. Aucun d'espace de noms
   </autreelement>
   <encoreunelement>
       ... Espace de nom par défaut
   </encoreunelement>
</element>
```

L'élément `element` est dans l'espace de noms `http://www.masociete.com` alors que l'élément `autreelement` n'est plus dans un espace de noms. L'élément `encoreunelement` se trouve également dans l'espace de noms `http://www.masociete.com`, de par l'espace de noms de son parent.

Exercice 4

Utilisation des espaces de noms par défaut et avec préfixe

Il s'agit de créer un document `livre3.xml` sur la base de `livre1.xml` en respectant les points suivants :

- Mettez tous les éléments dans l'espace de noms `http://www.masociete.com` sans utiliser d'espace de noms par défaut.
- Mettez la deuxième section dans un espace de noms `http://www.monentreprise.com`.
- Mettez le dernier paragraphe du dernier chapitre de la dernière section sans espace de noms.

Application d'un esspace de noms sur un attribut

Les espaces de nom peuvent s'appliquer via un préfixe sur un attribut ou une valeur d'attribut. Cet emploi peut servir, par exemple, à introduire des directives sans changer de structure. Cela peut également servir à contourner la règle qui veut que l'on ne puisse pas avoir plusieurs fois un attribut de même nom sur une déclaration d'élément.

Exemple :

```
<livre xmlns:p="http://www.imprimeur.com" p:quantite="p:50lots">
   <papier type="p:A4"/>
</livre>
```

Dans cet exemple, nous avons qualifié l'attribut `quantité` ainsi que les valeurs d'attribut `50lots` et `A4`.

Il existe une autre utilisation d'un espace sur une valeur d'attribut : un espace de noms permet de lever l'ambiguïté sur une valeur d'attribut. Prenons, par exemple, la valeur 1 : est-ce une chaîne de caractères ? Un nombre binaire ? Hexadécimal ? Un décimal ? Une seconde ? L'espace de noms sur une valeur d'attribut est couramment employé dans les schémas W3C que nous aborderons dans un autre chapitre.

Exercice 5

Utilisation des espaces de noms sur des attributs

Nous supposons que le livre des exercices précédents est maintenant disponible en plusieurs langues (au moins en français et en anglais).

Proposez une méthode pour gérer tous les titres et paragraphes en plusieurs langues.

Créez un document `livre4.xml` à partir de `livre1.xml`

Exemples de documents XML avec espace de noms

Pour vous exercer, vous pourrez analyser, dans les exemples ci-dessous, l'appartenance des différents éléments à tel ou tel espace de noms.

Le format XLink

XLink (*XML Linking Language* : *http://www.w3.org/XML/Linking*) sert à définir des liens entre différents documents. Dans la pratique, XLink est peu employé car il est plus simple d'introduire ses propres attributs ou balises pour réaliser des liens.

```
<annuaire xmlns:xlink="http://www.w3.org/1999/xlink">
 <persone
  xlink:href="etudiant/etudiant62.xml"
  xlink:label="Louis"
  xlink:role="http://www.campus.fr/etudiant"
  xlink:title="Louis Henri"/>
</annuaire>
```

`xlink` est le préfixe retenu pour réaliser un lien. Ce lien est effectué par l'attribut `href` vers un autre document ; le `label` est le texte affiché pour le lien ; le `role` est la nature du lien (un étudiant) et `title` est le titre du lien.

Le format XHTML

XHTML (*eXtensible HyperText Markup Language* : *http://www.w3.org/TR/xhtml1/*) est la version XML du standard HTML.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr">
  <head>
    <title>Ma page</title>
  </head>
  <body>
    <p>Mon texte</p>
  </body>
</html>
```

Les éléments XHTML sont donc dans l'espace de noms `http://www.w3.org/1999/xhtml`. À noter que la présence de l'attribut `lang` avec ou sans préfixe `xml` n'est liée qu'à une problématique de compatibilité avec les navigateurs HTML. Le préfixe `xml` existe par défaut pour désigner un attribut propre au standard.

Autre exemple introduisant des éléments MathML :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <body>
    <p>Formule mathématique</p>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply> <log/>
        <logbase>
          <cn> 3 </cn>
        </logbase>
        <ci> x </ci>
      </apply>
   </math>
  </body>
</html>
```

Outre les éléments XHTML, nous trouvons des éléments MathML dont l'espace de noms est *http://www.w3.org/1998/Math/MathML*.

Correction des exercices

L'ensemble des exercices a été réalisé avec le logiciel EditiX (*http://www.editix.com/*). Une version d'évaluation de 30 jours est librement téléchargeable (*http://www.editix.com/download.html*).

Exercice 1

```
<?xml version="1.0" encoding="iso-8859-1"?>

<livre>
  <titre>Mon livre</titre>
  <auteurs>
    <auteur><nom>Brillant</nom><prenom>Alexandre</prenom></auteur>
    <auteur><nom>Briand</nom><prenom>Aristide</prenom></auteur>
  </auteurs>
  <sections>
    <section>
       <titre>Section 1</titre>
       <chapitres>
           <chapitre>
               <titre>Chapitre 1</titre>
               <paragraphes>
                       <paragraphe>Premier paragraphe</paragraphe>
                       <paragraphe>Deuxième paragraphe</paragraphe>
               </paragraphes>
           </chapitre>
           <chapitre>
               <titre>Chapitre 2</titre>
               <paragraphes>
                       <paragraphe>Premier paragraphe</paragraphe>
                       <paragraphe>Deuxième paragraphe</paragraphe>
               </paragraphes>
           </chapitre>
       </chapitres>
   </section>

   <section>
       <titre>Section 2</titre>
       <chapitres>
           <chapitre>
               <titre>Chapitre 1</titre>
               <paragraphes>
                       <paragraphe>Premier paragraphe</paragraphe>
                       <paragraphe>Deuxième paragraphe</paragraphe>
               </paragraphes>
           </chapitre>
           <chapitre>
               <titre>Chapitre 2</titre>
               <paragraphes>
```

```
                    <paragraphe>Premier paragraphe</paragraphe>
                    <paragraphe>Deuxième paragraphe</paragraphe>
                </paragraphes>
            </chapitre>
        </chapitres>
    </section>
  </sections>
</livre>
```

Nous avons fait le choix de créer des balises supplémentaires telles que `auteurs`, `sections`, `chapitres`, `paragraphes` pour éviter de mélanger des ensembles distincts, comme le titre. Cela présente l'avantage de créer des blocs homogènes (tels que les auteurs, les sections, les chapitres…).

Exercice 2

```
<?xml version="1.0" encoding="iso-8859-1"?>
<livre titre="Mon livre">
  <auteurs>
   <auteur nom="Brillant" prenom="Alexandre"/>
   <auteur nom="Briand" prenom="Aristide"/>
  </auteurs>
  <sections>
   <section titre="Section 1">
       <chapitre titre="Chapitre 1">
           <paragraphe>Premier paragraphe</paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
       </chapitre>
       <chapitre titre="Chapitre 2">
           <paragraphe>Premier paragraphe</paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
       </chapitre>
   </section>
   <section titre="Section 2">
       <chapitre titre="Chapitre 1">
           <paragraphe>Premier paragraphe</paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
       </chapitre>
       <chapitre titre="Chapitre 2">
           <paragraphe>Premier paragraphe</paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
       </chapitre>
   </section>
  </sections>
</livre>
```

Comme l'élément `titre` disparaît au profit de l'attribut, nous pouvons alléger notre structure en éliminant les blocs superflus, comme `chapitres` ou `paragraphes`.

Exercice 3

```
<livre>
…
       <chapitre titre="Chapitre1">
           <paragraphe>&lt;element id="10"&gt;&amp;gt;&lt;/element&gt;</paragraphe>
           <paragraphe><![CDATA[<element id="10">&gt;</element>]]></paragraphe>
       </chapitre>
   </section>
</livre>
```

Exercice 4

```
<p1:livre titre="Mon livre" xmlns:p1="http://www.masociete.com"
➥xmlns:p2="http://www.monentreprise.com">
<p1:auteurs>
    <p1:auteur nom="nom1" prenom="prenom1"/>
    <p1:auteur nom="nom2" prenom="prenom2"/>
</p1:auteurs>
<p1:sections>
    <p1:section titre="Section1">
        <p1:chapitre titre="Chapitre1">
           <p1:paragraphe>Premier paragraphe</p1:paragraphe>
           <p1:paragraphe>Deuxième paragraphe</p1:paragraphe>
        </p1:chapitre>
    </p1:section>
    <p2:section titre="Section2">
        <p2:chapitre titre="Chapitre1">
           <p2:paragraphe>Premier paragraphe</p2:paragraphe>
           <p2:paragraphe>Deuxième paragraphe</p2:paragraphe>
        </p2:chapitre>
        <p2:chapitre titre="Chapitre2">
           <p2:paragraphe>Premier paragraphe</p2:paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
        </p2:chapitre>
    </p2:section>
</p1:sections>
</p1:livre>
```

Il y a plusieurs combinaisons possibles en fonction de l'utilisation du préfixe ou de l'espace de noms par défaut.

Exercice 5

```
<livre titre="Mon livre" en:titre="mybook" xmlns="francais" xmlns:fr2="francais"
➥xmlns:en="anglais">
<auteurs>
   <auteur nom="nom1" prenom="prenom1"/>
   <auteur nom="nom2" prenom="prenom2"/>
```

```
</auteurs>
<sections>
   <section fr2:titre="Section1" en:titre="Section1">
       <chapitre titre="Chapitre1" en:titre="Chapter1">
           <paragraphe>Premier paragraphe</paragraphe>
           <en:paragraphe>First paragraph</en:paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
           <en:paragraphe>Second paragraph</en:paragraphe>
       </chapitre>
   </section>
   <section titre="Section2" en:titre="Section2">
       <chapitre titre="Chapitre1" en:titre="Chapter1">
           <paragraphe>Premier paragraphe</paragraphe>
           <en:paragraphe>First paragraph</en:paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
           <en:paragraphe>Second paragraph</en:paragraphe>
       </chapitre>
       <chapitre titre="Chapitre2" en:titre="Chapter2">
           <paragraphe>Premier paragraphe</paragraphe>
           <en:paragraphe>First paragraph</en:paragraphe>
           <paragraphe>Deuxième paragraphe</paragraphe>
           <en:paragraphe>Second paragraph</en:paragraphe>
       </chapitre>
   </section>
</sections>
</livre>
```

Le choix d'utiliser un préfixe pour désigner une langue est discutable mais tout à fait opérationnel. Notre document garde toujours la même structure quelle que soit la langue et on peut déplacer une partie de l'ouvrage sans avoir à répéter l'opération pour chaque traduction.

3

Validation des documents XML

Ce chapitre vous explique comment valider un document XML. La validation est un moyen pour vérifier que votre document est conforme à une grammaire.

Rôle de la validation dans l'entreprise

Les échanges de données dans l'entreprise sont divers. Ils peuvent être liés aux infrastructures (réseaux, machines, automates, bases de données, logiciels…), aux métiers (échange entre personnes, échange entre services…), aux intervenants externes (délocalisation, fournisseurs…). Bref, toutes ces circulations de données participent au bon fonctionnement de l'entreprise. On peut faire l'analogie avec la circulation sanguine qui apporte aux différents organes les nutriments et l'oxygène indispensables.

Certaines données sont liées à une opération manuelle (saisie) et d'autres à une opération automatique (par exemple une exportation à partir d'une base de données). Dans les deux cas, on comprendra que des erreurs sont toujours possibles, soit par une erreur de saisie, soit par une erreur de programmation.

La validation va renforcer la qualité des échanges en contraignant l'émetteur de données et le consommateur de données à vérifier la cohérence des données structurées en XML. Par cohérence, il faut entendre à la fois le vocabulaire (éléments, attributs et espaces de noms) mais également, chose aussi importante, l'ordre et les quantités. En fin de compte, la validation revient à établir un visa sur le document XML.

La validation est donc importante. Il faut cependant en réserver l'usage à des documents qui présentent une certaine complexité et qui sont de taille raisonnable (moins de 100 Mo, par exemple). Lorsque la taille d'un document est trop élevée, la validation peut devenir très gourmande en ressources car, comme nous allons le voir, certaines règles nécessitent une vision globale du document et donc une présence en mémoire. L'idéal étant de vérifier les temps moyens, de contrôler si la réactivité dans un processus métier est suffisante. La validation, même si elle n'est pas toujours employée en production, peut toujours servir lors du développement à contrôler que les données XML sont correctement structurées. Dans un processus de développement en spirale (prototypages multiples qui tendent vers une version finale), les fluctuations des demandes client et les évolutions des programmes rendent indispensable la validation ne serait-ce que pour éviter d'éventuels problèmes de régression.

La plupart des outils, et notamment les parseurs XML, proposent des outils de validation. Les parseurs courants supportent une ou plusieurs formes de grammaires. Les DTD (*Document Type Definition*), étant la forme la plus ancienne, sont présentes dans la plupart des outils. Viennent ensuite ce qu'on nomme les schémas W3C, une forme de grammaire plus moderne mais également plus complexe. Enfin, il existe d'autres alternatives dont l'avenir est encore incertain, même si certains développeurs semblent déjà conquis par la simplicité de RelaxNG, par exemple.

La première forme de validation par DTD

Une DTD (*Document Type Definition*) est une forme de grammaire relativement ancienne car issue de l'univers SGML (*Standard Generalized Markup Language*). Elle a l'avantage d'être rapide à écrire, tout en présentant l'inconvénient d'être pauvre en possibilités de contrôle (typage de données, par exemple). Autre point négatif, les espaces de noms sont difficilement gérables car il faut intégrer, dans la grammaire, les préfixes, ce qui est contraire à l'esprit des espaces de noms.

Une DTD peut être interne ou externe au document XML. L'usage voudra que l'on privilégie la forme externe pour des raisons de maintenance et de facilité d'accès. Dans cette dernière forme, le parseur XML trouvera une référence dans chaque document XML vers la DTD externe par l'instruction d'en-tête DOCTYPE (voir le chapitre 2).

Par exemple, un document XML ayant une DTD externe `cours.dtd`, située dans le même répertoire que notre document XML (accès relatif), se présente sous la forme :

```
<?xml version="1.0"?>
<!DOCTYPE cours SYSTEM "cours.dtd">
<cours>
…
</cours>
```

Commençons maintenant à analyser la syntaxe d'une DTD. Tout d'abord, il est important de comprendre que cette syntaxe, même si elle est liée à un usage XML, n'est pas à base de balise mais à base d'instructions, selon la syntaxe `<!INSTRUCTION…>` (conséquence de la parenté avec SGML).

La définition d'un élément

L'élément (ou balise) est exprimé par l'instruction `ELEMENT` suivie du nom de l'élément que l'on souhaite décrire et de son contenu. Ce dernier n'englobe que les éléments situés directement sous cet élément (les éléments fils).

Voici une synthèse de cette syntaxe :

```
<!ELEMENT unNom DEF_CONTENU>
```

`DEF_CONTENU` peut contenir :

- `EMPTY` : l'élément n'a pas de contenu ; il est donc vide. Il peut cependant avoir des attributs.
- `ANY` : l'élément peut contenir n'importe quel élément présent dans la DTD.
- `(#PCDATA)` : l'élément contient du texte. Le caractère `#` est là pour éviter toute ambiguïté avec une balise et indique au parseur qu'il s'agit d'un mot-clé. `PCDATA` signifie *Parsable Character DATA*.
- Un élément placé entre parenthèses comme `(nom_element)`. Le nom d'un élément désigne une référence vers un élément décrit dans une autre partie de la DTD
- Un ensemble d'éléments séparés par des opérateurs, le tout placé entre parenthèses. L'opérateur de choix, représenté par le caractère `|`, indique que l'un ou l'autre de deux éléments (ou deux ensembles d'éléments) doit être présent. L'opérateur de suite (ou séquence), représenté par le caractère `,`, indique que les deux éléments (ou les deux ensembles d'éléments) doivent être présents. Des parenthèses supplémentaires peuvent être utilisées pour lever les ambiguïtés.

Remarques

Les mots-clés `EMPTY` et `ANY` s'emploient sans parenthèse. Les opérateurs de choix ou de séquence s'appellent également des connecteurs (ils connectent les éléments).

Quelques exemples :

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

Cela nous autorise deux documents XML, soit :

```
<personne>
  <nom_prenom>Brillant Alexandre</nom_prenom>
</personne>
```

ou bien :

```
<personne>
  <nom>Brillant</nom>
</personne>
```

Autre cas avec l'opérateur de séquence.

```
<!ELEMENT personne(prenom,nom)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT nom (#PCDATA)>
```

Ici, l'opérateur de séquence limite les possibilités à un seul document XML valide :

```
<personne>
  <prenom>Alexandre</prenom>
  <nom>Brillant</nom>
</personne>
```

Les contenus (élément ou groupe d'éléments) peuvent être quantifiés par les opérateurs `*`, `+` et `?`. Ces opérateurs sont liés au concept de cardinalité. Lorsqu'il n'y a pas d'opérateur, la quantification est de 1 (donc toujours présent).

Voici le détail de ces opérateurs :

- `*` : 0 à n fois ;
- `+` : 1 à n fois ;
- `?` : 0 ou 1 fois.

Quelques exemples :

```
<!ELEMENT plan (introduction?,chapitre+,conclusion?)>
```

L'élément `plan` contient un élément `introduction` optionnel, suivi d'au moins un élément `chapitre` et se termine par un élément `conclusion` optionnel également.

```
<!ELEMENT chapitre (auteur*,paragraphe+)>
```

L'élément `chapitre` contient de 0 à n éléments `auteur` suivi d'au moins un élément `paragraphe`.

```
<!ELEMENT livre (auteur?,chapitre)+>
```

L'élément `livre` contient au moins un élément, chaque élément, étant un groupe d'éléments où l'élément `auteur`, est optionnel et l'élément `chapitre` est présent en un seul exemplaire.

La définition d'un attribut

Les attributs sont précisés dans l'instruction `ATTLIST`. Cette dernière, étant indépendante de l'instruction `ELEMENT`, on précise à nouveau le nom de l'élément sur lequel s'applique le ou les attributs. On peut considérer qu'il existe cette forme syntaxique :

```
nom TYPE OBLIGATION VALEUR_PAR_DEFAUT
```

Le TYPE peut être principalement :

- CDATA : du texte (*Character Data*) ;
- ID : un identifiant unique (combinaison de chiffres et de lettres) ;
- IDREF : une référence vers un ID ;
- IDREFS : une liste de références vers des ID (séparation par un blanc) ;
- NMTOKEN : un mot (donc pas de blanc) ;
- NMTOKENS : une liste de mots (séparation par un blanc) ;
- Une énumération de valeurs : chaque valeur est séparée par le caractère |.

L'OBLIGATION ne concerne pas les énumérations qui sont suivies d'une valeur par défaut. Dans les autres cas, on l'exprime ainsi :

- #REQUIRED : attribut obligatoire.
- #IMPLIED : attribut optionnel.
- #FIXED : attribut toujours présent avec une valeur. Cela peut servir, par exemple, à imposer la présence d'un espace de noms.

La VALEUR_PAR_DEFAUT est présente pour l'énumération ou lorsque la valeur est typée avec #IMPLIED ou #FIXED.

Quelques exemples :

```
<!ATTLIST chapitre
 titre CDATA #REQUIRED
 auteur CDATA #IMPLIED>
```

L'élément chapitre possède ici un attribut titre obligatoire et un attribut auteur optionnel.

```
<!ATTLIST crayon
 couleur (rouge|vert|bleu) "bleu">
```

L'élément crayon possède un attribut couleur dont les valeurs font partie de l'ensemble rouge, vert, bleu.

La définition d'une entité

Les entités sont déclarées par l'instruction ENTITY. Comme nous l'avons abordé dans le chapitre précédent, l'entité associe un nom à une valeur. Ce nom est employé dans le document XML comme une forme d'alias ou de raccourci vers la valeur suivant la syntaxe &nom;. La valeur d'une entité peut être interne ou externe.

Dans la forme interne la syntaxe pour déclarer une entité est simplement la suivante :

```
<!ENTITY nom "VALEUR">
```

Dans la forme externe, on se retrouve avec le même principe qu'avec l'instruction DOCTYPE en tête du document XML assurant le lien vers une DTD. Les mots-clés `SYSTEM` et `PUBLIC` servent donc à réaliser un lien vers une valeur présente dans un fichier.

Exemple :

```
<!ENTITY nom SYSTEM "unTexte.txt">
```

L'entité `nom` est ici liée au contenu du fichier `unTexte.txt`.

Les entités ne s'appliquent pas uniquement au document XML. Elles peuvent également servir à la réalisation de la DTD pour limiter les répétitions de blocs de définition (par exemple, un attribut présent dans plusieurs éléments). Cette forme d'entité est appelée entité paramétrique et doit être déclarée suivant la syntaxe :

```
<!ENTITY % nom "VALEUR">
```

L'instruction `%nom;` sert à utiliser une entité paramétrique dans la DTD.

> **Remarque**
> Attention à ne pas confondre les caractères `&` et `%`.

Exemple :

```
<!ENTITY % type_defaut "CDATA">
<!ATTLIST chapitre
titre %type_defaut; #REQUIRED>
```

Dans cet exemple, nous avons créé une entité paramétrique `type_defaut` qui est associée à un type (`CDATA`) pour un attribut. Cette valeur est ensuite employée pour définir le typage de l'attribut `titre` de l'élément `chapitre`.

Grâce aux entités paramétriques, il est également possible d'activer ou de désactiver des blocs de définition. Ces blocs suivent la syntaxe suivante :

```
<![Valeur[
Partie de DTD
]]>
```

Si `Valeur` vaut `INCLUDE` alors la partie de DTD est activée. Si `Valeur` vaut `IGNORE` cette partie est ignorée.

Exemple :

```
<!ENTITY % anglais 'INCLUDE'>
<![%anglais;[
<!ATTLIST chapitre
  langue (anglais|français) "français">
]]>
```

Dans cet exemple, on définit un attribut `langue` pour un élément `chapitre` uniquement si l'entité paramétrique `anglais` a la valeur `INCLUDE`.

Remarque

Il est possible d'utiliser plusieurs instructions `ATTLIST` pour un même élément, le parseur effectuant la synthèse de tous les attributs définis.

Exercice 1

Utilisation d'une DTD

Créez la DTD `carnet.dtd` suivante :

```
<!ELEMENT carnet (personne+)>
<!ELEMENT personne EMPTY>
<!ATTLIST personne
nom CDATA #REQUIRED
prenom CDATA #IMPLIED
telephone CDATA #REQUIRED>
```

Créez un document XML qui soit valide par rapport à cette DTD.

Exercice 2

Création d'une DTD

Créez une DTD `livre.dtd` à partir du document `livre2.xml` créé dans le chapitre précédent.

Exercice 3

Utilisation des entités paramétriques

Modifiez la DTD créée dans l'exercice 2 pour faire en sorte que la définition de l'attribut `titre` soit unique à l'aide d'une entité paramétrique.

La validation par un schéma W3C

Un schéma W3C (que l'on nommera par la suite schéma) est une grammaire définie dans un formalisme XML. Sauf pour la gestion des entités, on peut considérer les schémas comme remplaçant les DTD. La version officielle est la 1.0 mais une version 1.1 est en préparation au moment de la rédaction de cet ouvrage. Quelques caractéristiques des schémas :

- gestion des espaces de noms ;
- types de base riches et extensibles ;

- réutilisation par importation et héritage ;
- davantage de souplesse dans les cardinalités ;
- proximité avec les schémas XDR (*XML-Data Reduced*).

Les différentes formes de type

Dans un schéma, les types sont fondamentaux. Ils ont la caractéristique d'être disponibles à la fois pour les éléments et les attributs.

Examinons, tout d'abord, le typage des éléments. Pour cela, passons en revue les contenus possibles pour un élément (que l'on nomme parfois modèle de contenu) :

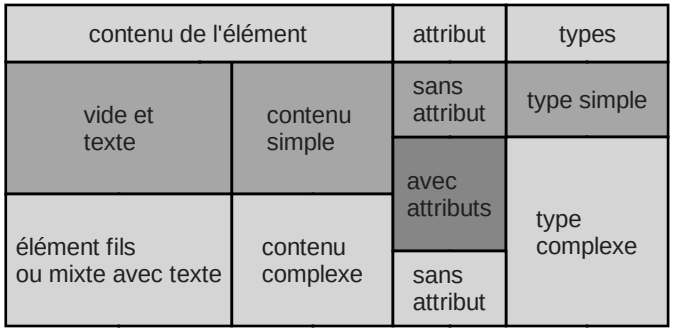
- Vide (entraîne `EMPTY` avec une DTD) : pas de contenu.
- Simple (entraîne `(#PCDATA)` avec une DTD) : du texte.
- Complexe : un ou plusieurs éléments.
- Mixte : un mélange d'éléments et de texte.

À partir de ces contenus possibles, il ressort dans les schémas 2 types :

- simple : du texte pour attribut ou élément ;
- complexe : avec au moins un attribut ou un élément fils.

> **Remarque**
> Le typage complexe va également s'appliquer dans le cas d'un contenu simple mais avec au moins un attribut. Il ne faut donc pas faire la confusion entre contenu simple et type simple.

Figure 3-1
Contenu d'un élément et typage



La figure 3-1 reprend le parallèle entre contenu d'un élément et typage. On le voit bien, la présence d'attributs déclenche nécessairement le passage en typage complexe.

Examinons maintenant le typage des attributs : il s'agit d'un typage simple qui suit donc le même principe que le typage d'un élément ne contenant que du texte et sans attributs.

Pour résumer :

- Pas de type :
  - élément sans contenu et sans attribut.
- Type simple :
  - Attribut ;
  - élément avec du texte seulement.
- Type complexe :
  - tous les autres cas d'usage de l'élément.

Les définitions globales et locales

Dans une DTD toutes les définitions sont globales. Cela signifie qu'il n'est pas possible d'utiliser plusieurs définitions pour un même nom d'élément.

L'exemple suivant ne peut pas être géré avec une DTD :

```
<document>
    <titre>…</titre>
<auteur>
    <titre>…</titre>
</auteur>
</document>
```

On voit bien que la balise `titre` n'a pas le même sens pour le document ou pour l'auteur. Pour réaliser une DTD, il faudrait décrire toutes les possibilités de contenu pour l'élément `titre`, avec la conséquence que l'on ne pourrait contrôler l'usage de telle ou telle possibilité (par exemple, le `titre` de l'auteur employé à la place du `titre` du document).

Dans un schéma, nous avons le choix, soit de créer une définition globale, soit de créer une définition locale (donc contextuelle à un élément parent). Une définition globale concerne également d'autres parties du schéma, comme les types, groupes, attributs…

L'assignation d'un schéma à un document XML

Le document XML est considéré comme une instance du schéma. Ce terme emprunte au monde objet certaines caractéristiques que nous aborderons ultérieurement.

Pour que le parseur puisse utiliser un schéma au cours de la validation, on insère dans le document XML des attributs supplémentaires sur l'élément racine. Ces attributs devront appartenir à l'espace de noms `http://www.w3.org/2001/XMLSchema-instance`. Il existe deux formes de liens :

- Le cas ignorant les espaces de noms du document. Dans ce cas, il faut utiliser l'attribut `noNamespaceSchemaLocation` pour désigner la localisation du schéma à employer. Cette localisation peut être relative au document XML ou absolue, contenir un chemin (*path*) ou une URL.

- Le cas prenant en compte les espaces de noms du document. Dans ce cas, on utilise l'attribut `schemaLocation`, qui a pour valeur une liste de couples (espace de noms et localisation d'un schéma). Nous aborderons cette syntaxe en détail de la gestion des espaces de noms.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<personnel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="personal.xsd">
…
</personnel>
```

Dans cet exemple, nous indiquons au parseur qu'il peut charger le schéma `personal.xsd` se trouvant dans le même répertoire que notre document XML.

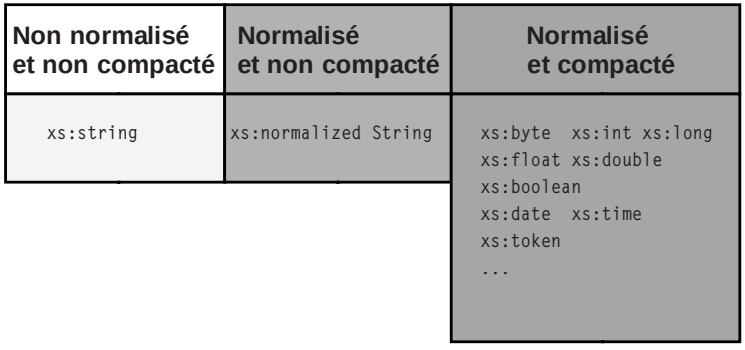
Les catégories de type simple

Les types applicables aux attributs et au contenu d'éléments simples (sans attribut) sont divisés en 2 catégories :

- Types normalisés : tous les caractères blancs (tabulation, retour chariot…) sont remplacés par un caractère unique.
- Types compactés : les occurrences de blancs en tête et queue sont supprimées. Les séries contiguës de blancs sont remplacées par un blanc.

Les types de base sont toujours liés à l'espace de noms des schémas pour éviter toute collision avec votre propre type. Lorsqu'on nomme un type de base, on le préfixe par `xs:` pour signifier la présence d'un espace de noms (et non pour obliger à utiliser le préfixe `xs`).

Figure 3-2
Répartition des types de base



La figure 3-2 représente quelques types de base que nous allons décrire dans la suite. On retiendra que le type `xs:string` est le type le plus permissif, dont on comprend aisément qu'il désigne une chaîne de caractères dont on souhaite conserver les blancs.

Voici un exemple d'utilisation de type simple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="titre" type="xs:string"/>
  <xs:element name="numchapitre" type="xs:int"/>
</xs:schema>
```

Sans trop rentrer dans les détails de la structure d'un schéma, vous observerez que le typage des éléments `titre` et `numchapitre` est lié à l'attribut `type`.

Voici deux documents possibles, valides par rapport à ce schéma :

```
<titre>Un titre</titre>
```

et

```
<numchapitre>10</numchapitre>
```

Les types pour les chaînes de caractères

Les types disponibles pour les chaînes de caractères sont les suivants :

- `xs:string` : le plus simple sans normalisation ni compactage ;
- `xs:normalizedString` : la forme normalisée et non compactée ;
- `xs:token` : la forme normalisée et compactée ;
- `xs:language` : les codes de langue (RFC 1766) ;
- `xs:NMTOKEN` : pas de blanc (comme `1999-10-10`, `1234534` ou `employe1023`) ;
- `xs:Name` : forme de `NMToken` commençant par une lettre ;
- `xs:ID` : chaîne unique dans le document ;
- `xs:IDREF` : chaîne ayant pour valeur un ID (d'où le mot `REF` pour référence) ;
- `xs:anyURI` : la valeur de l'URI utilisera des caractères ASCII, conversion automatique. Par exemple, `http://www.site.com/mon Rep` devient `http://www.site.com/mon%20Rep`.

Les types pour les dates et heures

Les dates et heures sont un sous-ensemble du standard ISO 8601 et s'appuient sur le calendrier grégorien (occidental). Les fuseaux horaires sont pris en compte par décalage avec le temps universel (TU). Cependant, les heures d'été et d'hiver ne sont pas gérées, ce dont il faudra tenir compte dans vos applications pour les comparaisons d'heure ou de date. Les types disponibles sont les suivants :

- Types : `xs:dateTime`, `xs:date`, `xs:time`, `xs:duration`, `xs:gYearMonth`, `xs:gYear`, `xs:gDay`, `xs:gMonth` ;
- Format lexical complet de `xs:dateTime` : `CCYY-MM-DDThh:mm:ssZff`.

Quelques exemples du type `xs:dateTime` :

- `2006-08-21T12:07:00` : fuseau horaire inconnu ;
- `2006-08-21T12:07:00+02:00` : décalage de 2 heures avec le TU ;
- `2006-08-21T12:07:00Z2` : décalage de 2 heures avec le TU.

Les types numériques

Les différents types numériques disponibles sont les suivants :

- Le plus simple `xs:int` (32 bits) : entier, également `xs:unsignedInt`.
- `xs:long` (64 bits), `xs:short` (16 bits), `xs:byte` (8 bits) : entier ; il existe également `xs:unsignedLong`, `xs:unsignedShort`, `xs:unsignedByte`.
- `xs:float` (32 bits) : flottant simple précision ; `xs:double` (64 bits) : flottant double précision. Quelques exemples : `3.14E10`, `3.14`, `4.14E-10`, `INF`, `-INF`, `NaN`.
- `xs:decimal` : nombre sans limitation ; le point est le séparateur décimal. Quelques exemples : `3.14`, `+3.14`, `-3.14`.
- `xs:integer` : forme de `xs:decimal` sans la partie décimale.
- `xs:nonPositiveInteger` : entier négatif avec le zéro ; `xs:negativeInteger` : entier négatif sans le zéro.
- `xs:boolean` : `true` ou `false` ou `1` ou `0`.

La création de nouveaux types simples

Les types simples peuvent être étendus pour créer de nouveaux types. Il existe trois possibilités de création :

- La restriction : comme son nom l'indique, on créé un sous-type en limitant certaines caractéristiques (par exemple, la longueur d'une chaîne). Les limitations sont définies avec des facettes.
- L'union : l'union sert à autoriser plusieurs types, ce qui permet d'offrir des alternatives, comme un attribut ayant pour valeur soit un entier soit une constante.
- La liste : il s'agit d'un ensemble de valeurs de même type séparées par un blanc. Il est également possible d'appliquer une restriction sur une liste, notamment pour en borner la taille.

Une première forme de création : la dérivation par restriction de type simple

Une restriction fait intervenir des balises qui vont réduire l'espace des valeurs possibles pour un type de base. Ces balises ont toutes un attribut `value` pour définir l'ampleur de la limitation.

La facette `whiteSpace` agit sur les chaînes de caractères. Elle peut prendre les valeurs `reserve` (blancs conservés), `replace` (blancs normalisés) ou bien `collapse` (occurrences de blanc ramenées à 1 blanc `&x20`).

Exemple :

```
<xs:simpleType name="chaineCompacte">
  <xs:restriction base="xs:String">
   <xs:whiteSpace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, nous avons créé le type `chaineCompacte` qui ne conserve pas les occurrences contiguës de blancs.

La facette `pattern` agit également sur les chaînes de caractères en vérifiant que la valeur est compatible avec une expression régulière. Plusieurs facettes peuvent être utilisées et, dans ce cas, il suffira que la valeur concorde avec au moins l'une des expressions régulières définies. Si vous n'êtes pas familier avec les expressions régulières, voici un rapide résumé.

Les expressions régulières (concept lié au *pattern matching*) représentent un langage pour décrire des mots (motifs) d'un point de vue syntaxique. Elles sont présentes dans la plupart des langages de programmation (Java, Perl...).

Voici une manière simple de les employer pour reconnaître un motif toujours de même forme :

```
<xs:simpleType nom="monEntier">
 <xs:restriction base="xs:int">
   <xs:pattern value="10"/>
   <xs:pattern value="30"/>
 </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, nous autorisons pour le type `monEntier`, ou bien l'entier `10` ou bien l'entier `30`. Cet exemple se rapproche d'une énumération.

Les caractères `{}`, `*`, `+` et `?` vont servir à exprimer des quantités de caractères :

- `{m,n}` signifie de `m` à `n` ;
- `*` signifie `0` ou plus ;
- `+` signifie au moins 1 ;
- `?` est l'option (`0` ou `1`).

Exemple :

```
<xs:simpleType>
 <xs:restriction base="xs:int">
  <xs:pattern value="0{1,2}10?"/>
 </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, nous autorisons les entiers `01`, `001`, `010` et `0010`.

Le langage des expressions régulières comprend également des classes de caractères :

- `\s` : blanc ;
- `\S` : tout sauf un blanc ;
- `\d` : un chiffre ;
- `\D` : tout sauf un chiffre ;
- `\w` : caractère alphanumérique plus "–" ;
- `\W` : tout sauf un caractère alphanumérique plus "–".

Exemple :

```
<xs:pattern value="\w\d+\w"/>
```

Cet exemple limite une valeur à un nombre alphanumérique de taille quelconque entouré de 2 caractères mot (`A123b`, `r1t`…).

Il est également possible de spécifier des plages de caractères avec les caractères crochets.

Exemples :

```
[0-9] : un chiffre
[a-Z] : une lettre minuscule
[a-zA-Z] : une lettre minuscule ou majuscule
[^0] : tout sauf 0
[0-9E] : un chiffre ou E
```

Exemple :

```
<xs:pattern value="[0-9]+[a-z]"/>
```

Ce dernier `pattern` comprend au moins un chiffre suivi d'un caractère alphabétique en minuscule (`1a`, `123b`, `99z`…).

Lorsqu'on souhaite reconnaître une valeur dans une liste de valeurs possibles on dispose, outre le `pattern`, de l'énumération (mot-clé `enumeration`).

Exemple :

```
<xs:simpleType name="couleurType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="bleu"/>
  <xs:enumeration value="blanc"/>
  <xs:enumeration value="rouge"/>
 </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, nous autorisons pour le type `couleurType` les valeurs `bleu`, `blanc`, `rouge`.

Remarque

Il est souvent agréable de terminer le nom d'un type par le suffixe `Type` afin d'être certain de ne pas faire de confusion avec d'autres noms (éléments, groupes…).

Voici un exemple complet reprenant notre type `couleurType` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="couleurType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="bleu"></xs:enumeration>
   <xs:enumeration value="blanc"></xs:enumeration>
   <xs:enumeration value="rouge"></xs:enumeration>
  </xs:restriction>
 </xs:simpleType>
 <xs:element name="crayon">
  <xs:complexType>
   <xs:attribute name="couleur" type="couleurType"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Pour les chaînes et nombres, nous disposons des facettes suivantes :

- Avec les chaînes :
  - `xs:length` : longueur en nombre de caractères ;
  - `xs:maxLength` : longueur maximale ;
  - `xs:minLength` : longueur minimale.
- Pour les nombres et les dates :
  - `xs:maxExclusive / xs:maxInclusive` : borne supérieure ;
  - `xs:minExclusive / xs:minInclusive` : borne inférieure.
- Pour les nombres seulement :
  - `xs:totalDigits` : nombre de chiffres d'un entier ;
  - `xs:fractionDigits` : nombre de chiffres après la virgule.

Autre forme de création : la construction de liste

L'autre forme de dérivation d'un type simple est la liste de valeurs. Les valeurs d'une liste sont séparées par un blanc et il est impossible d'utiliser un autre séparateur.

Premier exemple :

```
<xs:simpleType>
 <xs:list itemType="xs:int"/>
</xs:simpleType>
```

La liste `10 20 333 4 3` est donc bien cohérente avec ce type.

On peut écrire également une forme plus complexe en précisant de nouveaux types simples :

```
<xs:simpleType>
 <xs:list>
  <xs:simpleType>
   <xs:restriction base="xs:int">
```

```
    <xs:minInclusive value="40"/>
    <xs:maxInclusive value="50"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:list>
</xs:simpleType>
```

Dans cet exemple, nous autorisons une liste d'entiers bornés entre `40` et `50` (par exemple `40 43 50 44`).

Certaines facettes (`length`, `maxLength`, `minLength`, `enumeration`, `whiteSpace`) sont applicables aux listes.

Exemple :

```
<xs:simpleType name="maListeIniType">
 <xs:list itemType="xs:int"/>
</xs:simpleType>
<xs:simpleType name="maListeType">
 <xs:restriction base="maListeIniType">
  <xs:minLength value="5"/>
  <xs:maxLength value="6"/>
 </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, nous créons en premier lieu un type pour une liste d'entiers que nous avons appelé `maListeIniType`. Cette liste est ensuite restreinte par le type `maListeType` où nous bornons le nombre d'éléments de la liste de 5 à 6.

Dernière forme de construction de type simple : l'union

L'union sert à combiner plusieurs types simples ; il suffira qu'une valeur corresponde à l'un des types pour que la valeur soit correcte.

Exemple :

```
<xs:simpleType>
 <xs:union memberTypes="xs:int xs:boolean"/>
</xs:simpleType>
```

Dans cet exemple, on autorisera une valeur entière ou un booléen (`true` ou `false`).

Ce même exemple peut également être réécrit sous cette forme :

```
<xs:simpleType>
 <xs:union>
  <xs:simpleType>
   <xs:restriction base="xs:int"/>
  </xs:simpleType>
  <xs:simpleType>
   <xs:restriction base="xs:boolean"/>
  </xs:simpleType>
 </xs:union>
</xs:simpleType>
```

L'utilisation des types complexes

Pour rappel, un type complexe concerne le contenu d'un élément. Il caractérise une composition d'éléments ou d'attributs (d'où sa nature complexe). Dans un schéma, le type complexe nécessite toujours une balise `complexType`. Le type peut être global et donc associé à un nom ou bien être local à un élément.

Quelques exemples :

```
<auteur/> : pas un type complexe
<auteur>Mr Dupond</auteur> : pas un type complexe
<auteur id="10"/> : type complexe
<auteur id="10">Mr Dupond</auteur> : type complexe
<auteur><titre>Mr</titre><nom>Dupond</nom></auteur> : type complexe
```

Premier connecteur : la séquence

La séquence caractérise des éléments fils présents dans un ordre donné.

Exemple :

```
<xs:element name="plan">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="auteurs" type="xs:string"/>
   <xs:element name="chapitres" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Dans cet exemple, l'élément `plan` contient nécessairement 2 éléments fils de contenu simple `auteurs` et `chapitres`.

Nous aurions pu également l'écrire avec un type global (par exemple `planType`) :

```
<xs:complexType name="planType">
  <xs:sequence>
   <xs:element name="auteurs" type="xs:string"/>
   <xs:element name="chapitres" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
<xs:element name="plan" type="planType"/>
```

L'équivalent de cet exemple avec une DTD est :

```
<!ELEMENT plan (auteurs,chapitres)>
<!ELEMENT auteurs (#PCDATA)>
<!ELEMENT chapitres (#PCDATA)>
```

Deuxième connecteur : le choix

Le choix, comme son nom l'indique, étend les possibilités de constituer des documents XML valides en proposant des alternatives d'éléments.

Exemple :

```
<xs:element name="plan">
 <xs:complexType>
  <xs:choice>
   <xs:element name="sections" type="xs:string"/>
   <xs:element name="chapitres" type="xs:string"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

L'équivalent avec une DTD serait :

```
<!ELEMENT plan (sections|chapitres)>
<!ELEMENT sections (#PCDATA)>
<!ELEMENT chapitres (#PCDATA)>
```

Ce qui autorisera un élément `sections` ou bien un élément `chapitres` dans l'élément `plan`.

Dernier connecteur : tout

Le connecteur `all` est propre au schéma et caractérise un ensemble d'éléments de présence obligatoire mais sans contrainte sur l'ordre (toutes les permutations sont donc valides).

Exemple :

```
<xs:element name="plan">
 <xs:complexType>
  <xs:all>
   <xs:element name="auteur" type="xs:string"/>
   <xs:element name="chapitres" type="xs:string"/>
  </xs:all>
 </xs:complexType>
</xs:element>
```

Dans cet exemple l'élément `plan` contiendra les éléments `auteur` et `chapitres` dans n'importe quel ordre.

On peut simuler ce connecteur avec une DTD en écrivant ceci :

```
<!ELEMENT plan
 ((auteur,chapitres)|
  (chapitres,auteur))>
```

La limitation des quantités d'éléments : les cardinalités

Tout comme dans les DTD, les cardinalités sont possibles sur les éléments de connecteur ou sur les connecteurs eux-mêmes (jouant le rôle des parenthèses dans une DTD).

Ces cardinalités sont positionnées par les attributs `minOccurs` et `maxOccurs`. L'infini est caractérisé par la chaîne `unbounded`.

Pour faire le parallèle avec les DTD, nous retrouvons l'équivalent des opérateurs ?, + et * avec :

- ? : `minOccurs="0" maxOccurs="1"` ;
- + : `minOccurs="1" maxOccurs="unbounded"` ;
- * : `minOccurs="0" maxOccurs="unbounded"`.

Lorsqu'on ne précise rien, `minOccurs` et `maxOccurs` ont la valeur 1.

Exemple sur un élément :

```
<xs:element name="plan">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="auteur" type="xs:string"
maxOccurs="unbounded"/>
   <xs:element name="chapitre" type="xs:string"
minOccurs="2" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Dans cet exemple, l'élément `plan` contient un élément `auteur` suivi d'au moins 2 éléments `chapitre`.

Exemple sur un connecteur :

```
<xs:element name="plan">
 <xs:complexType>
   <xs:sequence minOccurs="2" maxOccurs="3">
   <xs:element name="auteur" type="xs:string"/>
   <xs:element name="chapitre" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Avec cet exemple, nous contrôlons que l'élément `plan` contient entre 2 et 3 suites d'éléments `auteur` et `chapitre`.

Remarque

Positionner des cardinalités sur des connecteurs a un effet multiplicatif sur le nombre d'éléments obligatoires ; ce n'est pas équivalent au fait de placer ces cardinalités sur les éléments contenus (comparez les 2 exemples précédents si vous n'êtes pas convaincu et déplacez les cardinalités sur la séquence ou les éléments).

Définition d'un attribut dans un type complexe

L'attribut implique la présence d'un type complexe. Il est toujours placé en dernière position. L'attribut en lui-même, ne contenant que du texte, est un type simple. L'attribut peut être global et donc réutilisable au sein de plusieurs définitions de type complexe.

Exemple :

```
<xs:element name="personne"
 <xs:complexType>
  …
  <xs:attribute name="nom" type="xs:string"/>
 </xs:complexType>
</xs:element>
```

Dans cet exemple, nous associons à l'élément `personne` l'attribut `nom`. L'attribut `nom` est local à l'élément `personne` ; il ne peut pas être employé dans un autre élément.

Pour créer un attribut réutilisable pour des définitions de type complexe, il faut le rendre global en le positionnant sous la racine `schema`. L'attribut `ref` sert à désigner la définition d'un attribut global.

Exemple :

```
<xs:schema …>
 <xs:attribute name="nom">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:minLength value="5"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <complexType name="monType">
  <xs:attribute ref="nom"/>
 </complexeType>
</xs:schema>
```

Dans cet exemple, l'attribut `nom` est déclaré globalement et est utilisé dans la définition d'un type complexe `monType`.

Limitation de l'attribut : les cardinalités

La présence d'un attribut peut être définie par l'attribut `use`, qui peut prendre les valeurs suivantes :

- `prohibited` : interdire l'usage d'un attribut par dérivation d'un type complexe.
- `optional` : l'attribut n'est pas obligatoirement renseigné (employé par défaut).
- `required` : l'attribut est obligatoire.

Deux autres attributs, `default` et `fixed`, servent à définir respectivement une valeur par défaut, si l'attribut est optionnel, et une valeur obligatoire.

Voici un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="a">
 <xs:complexType>
  <xs:attribute name="t1" use="required" type="xs:int"/>
  <xs:attribute name="t2" use="optional" type="xs:string" default="valeur"/>
```

```
  <xs:attribute name="t3" use="required" type="xs:token" fixed="autre"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Supposons qu'un document XML contienne l'extrait suivant :

```
<a t3="autre" t1="10"/>
```

Le parseur associera à cet élément les attributs et les valeurs suivantes : `t3=autre`, `t1=10`, `t2=valeur`.

Dans l'extrait suivant :

```
<a t3="autre" t1="11" t2="unevaleur"/>
```

le parseur associera à cet élément les attributs et les valeurs suivantes : `t3=autre`, `t1=11`, `t2=unevaleur`.

Les groupes d'attributs

Des définitions d'attributs communes à plusieurs définitions d'éléments peuvent être concentrées dans des groupes d'attributs. Les groupes d'attributs sont définis globalement et sont utilisés par référence.

Exemple :

```
<xs:attributeGroup name="RGB">
 <xs:attribute name="rouge" type="xs:byte"  use="required"/>
 <xs:attribute name="vert" type="xs:byte"  use="required"/>
 <xs:attribute name="bleu" type="xs:byte"  use="required"/>
</xs:attributeGroup>
```

Le groupe RGB contient la définition des trois attributs `rouge`, `vert` et `bleu`. Pour faire référence à ce groupe d'attributs, il suffit d'insérer l'instruction `<xs:attributeGroup ref="RGB"/>` à l'endroit où nous souhaitons utiliser ces trois attributs.

Les définitions d'éléments

Nous reprenons ici la correspondance entre le modèle de contenu d'un élément et sa validation dans un schéma. Vous serez ainsi en mesure de retrouver plus simplement la bonne définition.

La représentation de l'élément vide ou de contenu simple

L'élément vide n'a pas de contenu ; sa définition est donc sans typage. On l'écrit, par exemple :

```
<xs:element name="br"/>
```

La balise `br` ne peut donc s'exprimer que sous cette forme `<br/>`. L'inverse n'est malheureusement pas vrai : une balise vide ne signifie pas qu'il n'y a pas de type. Par exemple, le type `xs:string` accepte les chaînes vides.

L'élément de contenu simple sans attribut sera exprimé ainsi :

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

La balise `auteur` peut donc contenir une chaîne de caractères.

La représentation de l'élément vide avec attributs

L'attribut rend la structure de l'élément plus complexe. Son typage est donc également sous cette forme.

Voici un exemple :

```
<xs:element name="img">
 <xs:complexType>
  <xs:attribute name="src" type="xs:string"/>
 </xs:complexType>
</xs:element>
```

Cette balise s'exprimera donc dans un document XML sous cette forme :

```
<img src="..."/>
```

La représentation de l'élément avec contenu simple et attributs

Ce cas n'est pas forcément évident car il met en commun 2 principes antinomiques : le contenu est du texte que l'on peut définir par un type simple (par exemple `xs:string`), mais un typage complexe est également nécessaire, puisque l'élément contient un attribut (voir le début de chapitre sur les modèles de contenu, si cette notion ne vous est pas familière).

Les auteurs des schémas ont considéré que le passage d'un contenu simple à un type complexe pouvait se faire par dérivation. On l'exprimera par exemple ainsi :

```
<xs:element name="auteur">
 <xs:complexType>
  <xs:simpleContent>
   <xs:extension base="xs:string">
    <xs:attribut name="nom" type="xs:string"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

On peut le lire de la façon suivante : l'élément `auteur` est un type complexe dont le contenu simple (typé `xs:string`) a été étendu pour lui ajouter un attribut `nom`.

La représentation de l'élément avec contenu complexe et attributs

Ce cas est plus homogène, puisque le contenu complexe et l'attribut imposent un typage complexe.

Prenons l'exemple suivant :

```
<xs:element name="auteur">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="nom" type="xs:string"/>
   <xs:element name="prenom" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:token"/>
 <xs:complexType>
</xs:element>
```

Dans ce cas, l'élément `auteur` contient un élément `nom` suivi d'un élément `prenom` et possède un attribut `id`.

La représentation d'un contenu mixte

Un contenu mixte sert à faire cohabiter du texte et des éléments. On positionne un attribut `mixed` dans un type complexe pour obtenir l'effet recherché.

Un exemple :

```
<xs:complexType name="personType" mixed="true">
 <xs:sequence>
  <xs:element name="nom" type="xs:string"/>
  <xs:element name="prenom" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:element  name="employe" type="personType"/>
```

On pourrait donc, par exemple, écrire ce document XML :

```
<employe>Bonjour <nom>MrDupont</nom>,<prenom>Jean</prenom> de Paris</employe>
```

Remarque

Le texte du contenu mixte ne peut pas être typé (en dehors de `xs:string`) car il n'y aurait alors aucun sens à passer par de nouvelles balises pour préciser un contenu.

Limitation des schémas : le non-déterminisme

La conception d'un schéma ne doit pas entraîner de non-déterminisme, c'est-à-dire qu'il ne faut pas qu'à un endroit d'un document XML plusieurs définitions puissent être appliquées.

Un exemple simple :

```
<xs:complexType name="TYPE">
 <xs:choice>
  <xs:element name="A" type="TA"/>
   <xs:sequence>
    <xs:element name="A" type="TA"/>
```

```
    <xs:element name="C" type="TC"/>
   <xs:sequence>
  </xs:choice>
<xs:compleType>
```

Ce type fait intervenir 2 fois l'élément `A`, soit seul, soit suivi d'un élément `C`. Pour lever l'ambiguïté, il suffit d'écrire que l'élément `A` peut être suivi d'un élément `C` optionnel.

Réutilisation des définitions

Dans la réalisation d'un schéma, les économies de définitions sont d'autant plus importantes que certaines structures peuvent être complexes. Nous allons voir ici comment alléger vos déclarations.

Cas des définitions globales

Comme nous l'avons vu avec les attributs et les groupes d'attributs, une définition globale peut être employée à plusieurs endroits. On peut faire l'analogie dans un langage informatique avec une variable globale ou bien une classe publique (programmation objet).

Exemple avec un type complexe défini globalement :

```
<xs:complexType name="TYPE">
…
</xs:complexeType>
<xs:element name="A" type="TYPE"/>
<xs:element name="B" type="TYPE"/>
```

`TYPE` est un type complexe réemployable pour les éléments `A` et `B`.

Exemple avec une référence sur un élément :

```
<complexType>
 <xs:sequence>
  <xs:element ref="A"/>
 </xs:sequence>
</complexType>
```

Dans cette partie de définition, nous avons une séquence avec un élément `A` qui est lui-même défini globalement. Cet élément `A` peut donc se retrouver dans plusieurs définitions de contenu. Nous verrons également par la suite que cela a d'autres impacts.

Cas des groupes

Les groupes sont des agglomérats d'éléments. Ils servent avant tout à stocker des relations entre éléments qui peuvent être présentes dans plusieurs types complexes. On peut, par exemple, imaginer que les éléments `auteur` et `version` pourraient se retrouver dans différentes parties d'un document (livre, sections, chapitres…).

Un exemple :

```
<xs:group name="monGroupe">
 <xs:sequence>
  <xs:element name="contact" type="xs:string"/>
  <xs:element name="note" type="xs:string"/>
 </xs:sequence>
</xs:group>
<xs:element name="liste1">
 <xs:complexType maxOccurs="unbounded">
  <xs:group ref="monGroupe"/>
 </xs:complexType>
</xs:element>
```

Nous avons globalement défini le groupe `monGroupe` qui a été intégré à la définition complexe de l'élément `liste1`.

Exercice 4

Réalisation d'un schéma

Soit un document XML contenant un nombre indéterminé d'éléments sous la forme :

```
<contact titre="..." techno="...">
    <nom>...</nom>
    <prenom>...</prenom>
    <telephone> ...</telephone>
    <email>...</email>
    <email>...</email>
    ...
</contact>
```

L'élément `telephone` et l'attribut `techno` sont en option . Les textes seront des chaînes simples `xs:string`.

Vous utiliserez les types complexes `numerosType` et `contactType` pour construire un schéma nommé `annuaire.xsd`.

Exercice 5

Construction de types simples

Créez un schéma `annuaire2.xsd` à partir du schéma de l'exercice précédent.

Définissez et utilisez les types simples suivants :

- `technoType` : énumération dont les valeurs possibles sont `XML`, `Java`, `Autre`.
- `telType` : liste de 5 entiers (attention : créez d'abord un type pour la liste d'entiers).
- `emailType` : pattern `[a-z]+@[a-z]+\.[a-z]{2,3}`

Validez ce nouveau schéma sur un document de votre conception.

Exercice 6

Types complexes et groupes

Réalisez le schéma `livre.xsd` pour le document `livre2.xml` (construit dans le chapitre précédent).

Consignes :

1. N'utilisez pas de type complexe anonyme.

2. Créez et utilisez un groupe représentant une liste d'auteurs (`auteursGrp`).

3. Créez et utilisez un groupe d'attributs (`avecTitre`) représentant un titre.

4. Faites en sorte que chaque section puisse également contenir une liste d'auteurs (donc en utilisant le groupe).

Cas de l'héritage de contenu

Nous allons examiner ici les différentes formes d'héritage. Les concepteurs de schémas ont décelé des formes qui vont étendre des espaces de valeurs et d'autres qui vont les réduire.

L'héritage par extension

Nous avons vu jusqu'ici que le typage d'un contenu simple avec un attribut nécessitait une dérivation sous cette forme :

```
<xs:complexType>
 <xs:simpleContent>
  <xs:extension base="TYPE_SIMPLE">
   <xs:attribut name="MON_ATT" type="..."/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
```

Cette dérivation d'un contenu simple peut également exister avec un contenu complexe, comme :

```
<xs:complexType>
 <xs:complexContent>
   <xs:extension base="TC">
    <xs:sequence>
     <xs:element name="AJOUT" type="xs:string"/>
    </xs:sequence>
   </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Ici, nous avons ajouté l'élément `AJOUT` à un type complexe `TC`.

Prenons un exemple plus sophistiqué :

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

<xs:complexType name="employeType">
 <xs:complexContent>
  <xs:extension base="personType">
   <xs:sequence>
    <xs:element name="service" type="xs:string"/>
   </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="employe" type="employeType"/>
```

Le type global `personType` contient un élément `nom`. Le type complexe `employeType` étend le type `personType` en lui ajoutant l'élément `service`.

On peut donc écrire le document XML suivant :

```
<employe
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation
➥="monschema.xsd">
 <nom>Mr Dupont</nom>
 <service>Poste</service>
</employe>
```

La dérivation ne fonctionne pas n'importe comment. Elle entraînerait la présence d'une séquence même si le connecteur englobant le type de base était un choix.

Exemple :

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

<xs:complexType name="employeType">
<xs:complexContent>
  <xs:extension base="personType">
   <xs:choice>
    <xs:element name="service" type="xs:string"/>
   </xs:choice>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
```

En réalité, le type `employeType` est équivalent à :

```
<xs:sequence>
<xs:choice>
    <xs:element name="nom".../>
</xs:choice>
<xs:choice>
    <xs:element name="service".../>
</xs:choice
</xs:sequence>
```

L'héritage par restriction

Dans la restriction, on réduit l'espace des valeurs possibles. Cela fonctionne à la fois pour les contenus simples et pour les contenus complexes.

Prenons l'exemple d'un contenu simple :

```
<xs:complexType name="personType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="nom" type="xs:string"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="employeType">
<xs:simpleContent>
  <xs:restriction base="personType">
   <xs:length value="10"></xs:length>
  </xs:restriction>
</xs:simpleContent>
</xs:complexType>
<xs:element name="employe" type="employeType"/>
```

Le type complexe `personType` possède un attribut et un contenu simple. Nous avons réduit ce contenu simple par l'usage d'une facette dans le type complexe `employeType`.

La restriction peut également concerner les contenus complexes et, dans ce cas, il est impératif que la restriction reste compatible avec le type de base.

Voici un exemple :

```
<xs:complexType name="personType">
<xs:sequence>
  <xs:element name="nom" type="xs:string"/>
  <xs:element name="prenom" type="xs:string" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="employeType">
<xs:complexContent>
  <xs:restriction base="personType">
   <xs:sequence>
```

```
    <xs:element name="nom" type="xs:string"/>
   </xs:sequence>
  </xs:restriction>
</xs:complexContent>
</xs:complexType>
```

Dans ce cas de figure, le type complexe `personType` possède une séquence avec un élément `nom` et un élément `prenom` optionnel. La restriction définie dans `employeType` supprime l'élément `prenom` et reste donc cohérente avec le type `personType` qui rendait ce même élément optionnel.

L'héritage avec un contenu mixte

La dérivation d'un type complexe mixte entraîne un contenu complexe mixte ; autrement il y aurait une incohérence.

Par exemple :

```
<xs:complexType name="personType" mixed="true">
<xs:sequence>
 <xs:element name="nom" type="xs:string"/>
 <xs:element name="prenom" type="xs:string"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="employeType">
<xs:complexContent  mixed="true">
  <xs:extension base="personType">
   <xs:sequence>
    <xs:element name="service" type="xs:token"/>
   </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
```

Le type `employeType` dérive du type `personType` qui est mixte. Son contenu complexe est donc lui aussi mixte.

Exercice 7

Dérivation d'un type complexe

Créez un schéma `livre2.xsd` à partir du schéma `livre.xsd` élaboré dans l'exercice 6.

Créez un type complexe `avecTitreType` contenant l'attribut `titre`. Faites dériver tous les types avec cet attribut du type `avecTitreType`.

Testez votre nouveau schéma en validant `livre2.xml`.

L'utilisation des clés et références de clés

Analogie avec les DTD : ID et IDREF

Dans une DTD les types `ID` et `IDREF` appliqués aux attributs ont respectivement pour rôle d'imposer un identifiant unique et de faire référence à un identifiant présent dans le document (notion de lien).

Ces types sont à nouveau disponibles dans les schémas (`xs:ID` et `xs:IDREF`). Ils sont probablement à éviter, car ils offrent trop de limitations (pas de réel contrôle sur l'unicité et sur les références).

Un exemple :

```
<xs:element name="service">
<xs:complexType>
  <xs:sequence>
   <xs:element name="employe" type="xs:ID"  maxOccurs="unbounded"/>
   <xs:element name="bureau" type="xs:IDREFS"></xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
```

Un document XML valide pourrait être alors :

```
<service
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="monSchema.xsd">
 <employe>Dupont</employe>
 <employe>Doe</employe>
 <bureau>Dupont Doe</bureau>
</service>
```

La forme contextuelle : clé et unicité

Les contraintes d'unicité sont très proches de la notion d'identifiant. Une clé est une valeur unique qui peut être associée à une référence de clé. Une valeur unique n'est pas forcément une clé si elle n'a pas de valeur (et on ne peut donc pas y faire référence). C'est cette distinction que l'on retrouve dans nos schémas.

Les contraintes d'unicité ou de clé vont être délimitées par une expression XPath. Une expression XPath est une requête qui sert principalement à extraire telle ou telle partie de l'arborescence XML. Dans les schémas, les requêtes possibles ne sont qu'un sous-ensemble des possibilités du langage XPath. On peut réduire la notion de requête à la notion de chemin vers une partie du document, ce chemin étant similaire à un chemin vers un fichier, comme on en trouve sous Windows ou Linux/Unix.

Ces contraintes sont positionnées dans la définition d'un élément. Les requêtes XPath sont relatives à cet élément. Deux balises vont servir à localiser l'unicité ou la clé :

- `selector` : indique sur quels éléments est vérifiée la contrainte d'unicité ou de clé suivante.

- `field` : détermine où se situe la valeur qui doit être unique ou vide. Cette balise peut être répétée si l'unicité relève de contraintes multiples (plusieurs attributs…).

Exemple avec une contrainte d'unicité :

```
<xs:element name="service">
<xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element name="employe" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:unique name="e">
  <xs:selector xpath="employe"/>
  <xs:field xpath="."/>
</xs:unique>
</xs:element>
```

Dans cet exemple, nous demandons à ce que le contenu (texte) de tous les éléments `employe` présents dans l'élément `service` soit unique ou vide.

Remarque concernant l'expression XPath

Le point caractérise un contenu, on peut l'assimiler à un contenu texte. Il pourrait aussi se traduire par la fonction `text()`.

Voici un document XML qui respecte la contrainte précédente :

```
<service
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="monSchema.xsd">
 <employe>Dupont</employe>
 <employe>Doe</employe>
 <employe/>
</service>
```

Vous pourrez noter que le dernier élément `employe` n'a pas de contenu car il est assimilé à une chaîne vide.

Autre exemple avec une contrainte sur plusieurs attributs :

```
<xs:element name="service">
 <xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element name="employe">
    <xs:complexType>
     <xs:attribute name="nom" type="xs:string"/>
     <xs:attribute name="prenom" type="xs:string"/>
    </xs:complexType>
   </xs:element>
```

```
  </xs:sequence>
 </xs:complexType>
 <xs:unique name="e">
  <xs:selector xpath="employe"/>
  <xs:field xpath="@nom"/>
  <xs:field xpath="@prenom"/>
 </xs:unique>
</xs:element>
```

Ici, nous demandons à ce qu'il n'y ait pas de duplicata de valeur sur les attributs `nom` et `prenom` de l'élément `employe`.

Si nous utilisions maintenant une clé, nous obtiendrions quelque chose de similaire en remplaçant les éléments `unique` par `key`. Une clé doit aussi avoir un nom, car une référence de clé y est généralement associée.

Voici un exemple qui désigne un attribut `id` sur l'élément `employe` comme critère de clé.

```
<xs:element name="service">
<xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element name="employe">
    <xs:complexType>
     <xs:attribute name="nom" type="xs:string"/>
     <xs:attribute name="id" type="xs:token"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="cle">
  <xs:selector xpath="employe"/>
  <xs:field xpath="@id"/>
 </xs:key>
</xs:element>
```

Les liens avec les clés

La référence de clé s'exprime par l'élément `keyref` et doit se trouver au même niveau que la clé ou appartenir à la définition d'un élément ancêtre. L'attribut `refer` désigne à quelle clé on souhaite faire référence. La référence de clé s'exprime comme la clé en précisant les parties de l'arbre qui contiennent la valeur.

Voici un exemple :

```
<xs:element name="service">
 <xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element name="employe">
    <xs:complexType>
```

```
    <xs:attribute name="nom" type="xs:string"/>
     <xs:attribute name="id" type="xs:token"/>
     <xs:attribute name="chef" type="xs:token"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
 <xs:key name="kid">
  <xs:selector xpath="employe"/>
  <xs:field xpath="@id"/>
 </xs:key>
 <xs:keyref name="chef" refer="kid">
  <xs:selector xpath="employe"/>
  <xs:field xpath="@chef"/>
 </xs:keyref>
</xs:element>
```

Un extrait XML qui serait lié à cette association clé et référence de clé pourrait être :

```
<employe nom="dupont jean" id="E1"/>
<employe nom="dupont louis" id="E2" chef="E1"/>
```

Dans l'exemple suivant, les définitions de la clé et de la référence de clé sont présentes à des niveaux différents :

```
<xs:element name="service">
 <xs:complexType>
 ...
 </xs:complexType>
 <xs:key name="kid">
  <xs:selector xpath="employe"/>
  <xs:field xpath="@id"/>
 </xs:key>
</xs:element>
<xs:element name="entreprise">
 <xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element  ref="service"/>
  </xs:sequence>
 </xs:complexType>
 <xs:keyref name="chef" refer="kid">
  <xs:selector xpath="service/employe"/>
  <xs:field xpath="@chef"></xs:field>
 </xs:keyref>
</xs:element>
```

L'élément `entreprise` peut contenir une référence de clé vers l'attribut `id` de l'élément `employe`, car l'élément `entreprise` est ancêtre de l'élément `employe`.

Exercice 8

Clé et référence de clé

Créez le document `annuaire3.xsd` à partir du schéma `annuaire2.xsd` élaboré dans l'exercice 5.

Ajoutez les attributs `id` et `enRelation` à l'élément `contact`.

Employez les clés et les références de clés pour garantir l'unicité des `id` et l'usage de lien correct (par `enRelation`).

Créez un document XML et validez-le.

Aide : utilisez les expressions XPath : `contact`, `@id` et `@enRelation`. Placez les `key` et `keyRef` dans l'élément racine.

Relations entre schémas

Comme pour les définitions, les schémas peuvent, selon leur conception, être plus ou moins réutilisables. Nous allons décrire ici différents cas.

L'inclusion d'un schéma

Il existe différentes techniques pour agglomérer des schémas. La première est l'inclusion et consiste simplement à injecter un schéma dans un autre.

L'élément `include` qui permet de réaliser cette inclusion se positionne sous la racine du schéma.

Soit le schéma suivant `personne.xsd` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType  name="personneType">
  <xs:sequence>
   <xs:element name="nom" type="xs:string"/>
   <xs:element name="prenom" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Le schéma `employe.xsd` inclut le schéma `personne.xsd` de la façon suivante :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="personne.xsd"/>
 <xs:element name="employe" type="personneType"/>
</xs:schema>
```

Nous pourrions, par exemple, réaliser, à partir de ce dernier schéma, le document suivant :

```
<employe
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="employe.xsd">
 <nom>Doe</nom>
 <prenom>John</prenom>
</employe>
```

L'inclusion est assez neutre dans la gestion des espaces de noms. Le schéma inclus *épouse* l'espace de noms du schéma principal. Nous aborderons ces concepts quelques peu complexes dans le prochain chapitre.

L'inclusion et la redéfinition d'un schéma

La redéfinition améliore l'inclusion en offrant la possibilité de redéfinir un type simple ou complexe. On peut faire l'analogie avec une surcharge en programmation objet, où le contenu d'une méthode est réécrit dans une classe dérivée.

Comme nous avons déjà vu les notions de dérivation, voici un exemple qui illustre l'usage de la redéfinition avec l'élément `redefine`. Le schéma `employe.xsd` est le même que précédemment.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:redefine schemaLocation="employe.xsd">
 <xs:complexType name="personneType">
  <xs:complexContent>
    <xs:extension base="personneType">
     <xs:sequence>
      <xs:element name="telephone" type="xs:string"/>
     </xs:sequence>
    </xs:extension>
   </xs:complexContent>
 </xs:complexType>
</xs:redefine>
</xs:schema>
```

Dans cet exemple, nommé `employe2.xsd`, nous avons greffé un élément `telephone` à la définition d'une personne.

Nous pourrions donc maintenant écrire quelque chose dans ce style :

```
<employe
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
➥si:noNamespaceSchemaLocation="employe2.xsd">
 <nom>Doe</nom>
 <prenom>John</prenom>
 <telephone>999</telephone>
</employe>
```

Exercice 9

Inclusion de schéma

Créez un schéma `auteurs.xsd` contenant tous les types et groupes liés aux auteurs (faites un copier-coller à partir de `livre.xsd`).

Créez un schéma `livre3.xsd` à partir du schéma `livre.xsd` élaboré dans l'exercice 6 en éliminant les types et les groupes liés aux auteurs et en incluant `auteur.xsd`. Testez votre nouveau schéma dans un document XML.

Quels sont les avantages de l'inclusion ?

Documentation d'un schéma W3C

La documentation dans un schéma peut bien entendu être présente via de simples commentaires XML. Les schémas mettent cependant à disposition les balises `annotation` et `documentation`, ces balises se positionnant dans les parties que l'on souhaite commenter.

La documentation peut utiliser plusieurs formats et plusieurs langues.

Voici un exemple en plusieurs langues sur la balise `service` :

```
<xs:element name="service">
<xs:annotation>
  <xs:documentation xml:lang="fr">
  Service de l'entreprise</xs:documentation>
  <xs:documentation xml:lang="en">
  Service of the company</xs:documentation>
 </xs:annotation>
 ...
</xs:element>
```

Cet autre exemple utilise des balises XHTML :

```
<xs:annotation>
<xs:documentation xml:lang="fr">
    <p xmlns="http://www.w3.org/1999/xhtml">Service <b>de</b> l'entreprise</p>
 </xs:documentation>
</xs:annotation>
```

Conclusion sur les schémas

Les schémas restent complexes. Cependant, il est de plus en plus difficile de les éviter car le W3C les introduit à la fois comme garant des spécifications et dans certains langages comme XSLT 2.0, WSDL ou XForms. À noter que les schémas devraient être simplifiés dans les prochaines versions.

Nous aborderons dans un prochain chapitre d'autres concept reliés aux schémas, comme la gestion des espaces de noms ou diverses techniques de modélisation.

La validation avec le format RelaxNG

RelaxNG est une alternative séduisante au schéma W3C développée par James clark basé sur Relax et TREX (deux autres schémas). RelaxNG a deux formes, l'une en pur XML et l'autre sous une forme plus compacte (façon DTD). RelaxNG a l'avantage de gérer les espaces de noms et d'être relativement intuitif.

Voici un lien pour obtenir davantage d'information sur RelaxNG : *http://relaxng.org/*.

Les principaux inconvénients de RelaxNG sont :

- Pas de standardisation par le W3C.

- Encore trop peu d'outils/parseurs gérant RelaxNG.
- Pas d'éditeur graphique.
- Avenir assez incertain compte tenu des évolutions des schémas W3C.

Voici un exemple de grammaire RelaxNG et sa correspondance avec une DTD.

```
<element
name="addressBook"
xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

Voici la DTD correspondante :

```
<!ELEMENT addressBook (card*)>
<!ELEMENT card (name, email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

Correction des exercices

L'ensemble des exercices a été réalisé sur le logiciel EditiX (*http://www.editix.com/*). Une version d'évaluation de 30 jours est librement téléchargeable (*http://www.editix.com/download.html*).

Exercice 1

```
<?xml version="1.0"?>
<!DOCTYPE carnet SYSTEM "carnet.dtd">
<carnet>
 <personne nom="dupont" prenom="jean" telephone="001122"/>
 <personne nom="dupond" telephone="221100"/>
</carnet>
```

Exercice 2

```
<!ELEMENT livre (auteurs,sections)>
<!ELEMENT auteurs (auteur+)>
<!ELEMENT auteur EMPTY>
<!ELEMENT sections (section+)>
```

```
<!ELEMENT section (chapitre,chapitre+)>
<!ELEMENT chapitre (paragraphe,paragraphe+)>
<!ELEMENT paragraphe (#PCDATA)>

<!ATTLIST livre
titre CDATA #REQUIRED>
<!ATTLIST section
titre CDATA #REQUIRED>
<!ATTLIST chapitre
titre CDATA #REQUIRED>
<!ATTLIST auteur
nom CDATA #REQUIRED
prenom CDATA #REQUIRED>
```

Exercice 3

```
<!ELEMENT livre (auteurs,sections)>
<!ELEMENT auteurs (auteur+)>
<!ELEMENT auteur EMPTY>
<!ELEMENT sections (section+)>
<!ELEMENT section (chapitre,chapitre+)>
<!ELEMENT chapitre (paragraphe,paragraphe+)>
<!ELEMENT paragraphe (#PCDATA)>
<!ENTITY % titre "titre CDATA #REQUIRED">
<!ATTLIST livre
%titre;>
<!ATTLIST section
%titre;>
<!ATTLIST chapitre
%titre;>
<!ATTLIST auteur
nom CDATA #REQUIRED
prenom CDATA #REQUIRED>
```

Exercice 4

```
<?xml version="1.0" encoding="ISO-8859-1"?>

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

<xs:complexType name="numerosType">
 <xs:sequence>
  <xs:element name="contact" type="contactType" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="contactType">
 <xs:sequence>
  <xs:element name="nom" type="xs:string"/>
  <xs:element name="prenom" type="xs:string"/>
  <xs:element name="telephone" type="xs:string" minOccurs="0"/>
```

```
  <xs:element name="email" type="xs:string" maxOccurs="unbounded"/></xs:sequence>
 <xs:attribute name="titre" type="xs:string" use="required"/>
 <xs:attribute name="techno" type="xs:string" use="optional"/>
</xs:complexType>

<xs:element name="numeros" type="numerosType"/>
</xs:schema>
```

Exercice 5

```
<?xml version="1.0" encoding="ISO-8859-1"?>

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

<xs:simpleType name="technoType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="XML"/>
  <xs:enumeration value="Java"/>
  <xs:enumeration value="Autre"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="listIntType">
  <xs:list  itemType="xs:int">
  </xs:list>
 </xs:simpleType>

<xs:simpleType name="telType">
  <xs:restriction base="listIntType">
   <xs:length value="5"/>
  </xs:restriction>
 </xs:simpleType>

<xs:simpleType name="emailType">
  <xs:restriction base="xs:string">
   <xs:pattern value="[a-z]+@[a-z]+\.[a-z]{2,3}"/>
  </xs:restriction>
 </xs:simpleType>

<xs:complexType name="numerosType">
  <xs:sequence>
   <xs:element name="contact" type="contactType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="contactType">
  <xs:sequence>
   <xs:element name="nom" type="xs:string"/>
   <xs:element name="prenom" type="xs:string"/>
   <xs:element name="telephone" type="telType" minOccurs="0"/>
```

```
    <xs:element name="email" type="emailType" maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="titre" type="xs:string" use="required"/>
   <xs:attribute name="techno" type="technoType" use="optional"/>
  </xs:complexType>

  <xs:element name="numeros" type="numerosType"/>

 </xs:schema>
```

Exercice 6

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
attributeFormDefault="unqualified"
elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="livreType">
 <xs:sequence>
  <xs:element name="auteurs" type="auteursType"/>
  <xs:element name="sections" type="sectionsType"/>
 </xs:sequence>
 <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:complexType name="auteursType">
 <xs:group ref="auteursGrp"/>
</xs:complexType>
<xs:group name="auteursGrp">
 <xs:sequence>
  <xs:element maxOccurs="unbounded" name="auteur" type="auteurType"/>
 </xs:sequence>
</xs:group>
<xs:attributeGroup name="avecTitre">
 <xs:attribute name="titre" type="xs:string" use="required"/>
</xs:attributeGroup>
<xs:complexType name="auteurType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="nom" type="xs:string" use="required"/>
   <xs:attribute name="prenom" type="xs:string" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
<xs:complexType name="sectionsType">
 <xs:sequence>
  <xs:element maxOccurs="unbounded" minOccurs="2" name="section" type="sectionType"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="sectionType">
 <xs:sequence>
  <xs:group minOccurs="0" ref="auteursGrp"/>
<xs:element maxOccurs="unbounded" minOccurs="2" name="chapitre" type="chapitreType"/>
 </xs:sequence>
```

```
 <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:complexType name="chapitreType">
 <xs:sequence>
  <xs:element maxOccurs="unbounded" minOccurs="2" name="paragraphe" type="xs:string"/

 </xs:sequence>
 <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:element name="livre" type="livreType"/>
</xs:schema>
```

Exercice 7

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="avecTitreType">
 <xs:attribute name="titre" type="xs:string"/>
</xs:complexType>
<xs:complexType name="livreType">
<xs:complexContent>
  <xs:extension base="avecTitreType">
   <xs:sequence>
    <xs:element name="auteurs" type="auteursType"/>
    <xs:element name="sections" type="sectionsType"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="auteursType">
 <xs:group ref="auteursGrp"/>
</xs:complexType>
<xs:group name="auteursGrp">
 <xs:sequence>
  <xs:element maxOccurs="unbounded" name="auteur" type="auteurType"/>
 </xs:sequence>
</xs:group>
<xs:complexType name="auteurType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="nom" type="xs:string" use="required"/>
   <xs:attribute name="prenom" type="xs:string" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
<xs:complexType name="sectionsType">
 <xs:sequence>
  <xs:element maxOccurs="unbounded" minOccurs="2" name="section" type="sectionType"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="sectionType">
 <xs:complexContent>
```

```
  <xs:extension base="avecTitreType">
   <xs:sequence>
    <xs:group minOccurs="0" ref="auteursGrp"/>
    <xs:element maxOccurs="unbounded" minOccurs="2" name="chapitre"
    ➥type="chapitreType"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="chapitreType">
 <xs:complexContent>
  <xs:extension base="avecTitreType">
   <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="2" name="paragraphe" type="xs:string"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:element name="livre" type="livreType"/>
</xs:schema>
```

Exercice 8

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:simpleType name="listIntType">
 <xs:list itemType="xs:int"/>
</xs:simpleType>
<xs:simpleType name="telType">
 <xs:restriction base="listIntType">
  <xs:length value="5"/>
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="emailSansAttributType">
 <xs:restriction base="xs:string">
  <xs:pattern value="[a-z]+@[a-z]+\.[a-z]{2,3}"/>
 </xs:restriction>
</xs:simpleType>
<xs:complexType name="emailType">
 <xs:simpleContent>
  <xs:extension base="emailSansAttributType">
   <xs:attribute name="nature" use="required">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="perso"/>
      <xs:enumeration value="travail"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:extension>
 </xs:simpleContent>
```

```
</xs:complexType>
<xs:complexType name="numerosType">
 <xs:sequence>
   <xs:element maxOccurs="unbounded" name="contact" type="contactType"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="contactType">
 <xs:sequence>
  <xs:element name="nom" type="xs:string"/>
  <xs:element name="prenom" type="xs:string"/>
  <xs:element minOccurs="0" name="telephone" type="telType"/>
  <xs:element maxOccurs="unbounded" name="email" type="emailType"/>
 </xs:sequence>
 <xs:attribute name="titre" type="xs:string" use="required"/>
 <xs:attribute name="techno" use="optional">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="XML"/>
    <xs:enumeration value="Java"/>
    <xs:enumeration value="Autre"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="id" type="xs:string"/>
 <xs:attribute name="enRelation" type="xs:string"/>
</xs:complexType>
<xs:element name="numeros" type="numerosType">
 <xs:key name="id">
  <xs:selector xpath="contact"/>
  <xs:field xpath="@id"/>
 </xs:key>
 <xs:keyref name="refId" refer="id">
  <xs:selector xpath="contact"/>
  <xs:field xpath="@enRelation"/>
 </xs:keyref>
</xs:element>
</xs:schema>
```

Exercice 9

auteurs.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="auteursType">
 <xs:group ref="auteursGrp"/>
</xs:complexType>
<xs:group name="auteursGrp">
 <xs:sequence>
  <xs:element name="auteur" type="auteurType" maxOccurs="unbounded"/>
 </xs:sequence>
```

```
</xs:group>
<xs:complexType name="auteurType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="nom" type="xs:string" use="required"/>
   <xs:attribute name="prenom" type="xs:string" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
</xs:schema>
```

livre3.xsd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:include schemaLocation="auteurs.xsd"/>
<xs:complexType name="livreType">
    <xs:sequence>
        <xs:element name="auteurs" type="auteursType"/>
        <xs:element name="sections" type="sectionsType"/>
    </xs:sequence>
    <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:attributeGroup name="avecTitre">
    <xs:attribute name="titre" type="xs:string" use="required"/>
</xs:attributeGroup>
<xs:complexType name="sectionsType">
    <xs:sequence>
        <xs:element name="section" type="sectionType" minOccurs="2"
        ➥maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="sectionType">
    <xs:sequence>
        <xs:group ref="auteursGrp" minOccurs="0"/>
        <xs:element name="chapitre" type="chapitreType" minOccurs="2"
        ➥maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:complexType name="chapitreType">
    <xs:sequence>
        <xs:element name="paragraphe" type="xs:string" minOccurs="2"
        ➥maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attributeGroup ref="avecTitre"/>
</xs:complexType>
<xs:element name="livre" type="livreType"/>
</xs:schema>
```

4

Modélisation XML

Dans l'acte de modéliser, il y a la notion de représenter un concept pour mieux le manipuler. La modélisation XML peut être liée aux schémas qui autorisent ou interdisent l'utilisation de certaines formes XML. Nous aborderons, dans ce chapitre, d'autres représentations des schémas W3C, que ce soit à travers la gestion des espaces de noms, par analogie avec UML ou encore avec les design patterns *(patrons de conception).*

Modélisation avec les espaces de noms

Dans le chapitre précédent, nous avons vu comment construire un schéma, mais nous n'avons pas abordé la relation entre l'espace de noms et le schéma. Nous allons constater, dans ce chapitre, que les espaces de noms constituent un autre axe de relation entre schémas.

À un espace de noms, on attache un schéma ; à un schéma, on attache au plus un espace de noms (car on peut ne pas avoir d'espaces de noms associés). L'agglomération des espaces de noms (votre document XML est composé de plusieurs espaces de noms) est également définie dans un schéma principal. Seules les définitions (éléments, attributs…) restent externalisées.

Soit le document XML suivant :

```
<entreprise>
 <service xmlns="http://www.monentreprise.com/secretariat">
  <employe nom="dupond"/>
 </service>
```

```
 <direction xmlns="http://www.monentreprise.com/direction">
  <employe nom="doe"/>
 </direction>
</entreprise>
```

Nous avons trois cas d'utilisation des espaces de noms. Tout d'abord l'élément racine `entreprise` n'a pas d'espace de noms. L'élément `service` appartient à l'espace de noms `http://www.monentreprise.com/secretariat` alors que l'élément `direction` appartient à l'espace de noms `http://www.monentreprise.com/direction`. Nous aurons donc trois schémas à réaliser. Le premier schéma sera lié au cas sans espace de noms et fera référence aux deux autres schémas.

L'attribut targetNamespace

`targetNamespace` est un attribut que l'on positionne à la racine d'un schéma pour désigner l'espace de noms lié. Cet attribut a un impact sur toutes les définitions globales (éléments, attributs, types…), ces dernières devenant d'office qualifiées dans l'espace de noms de ce schéma. Cette règle ne s'applique pas aux définitions locales ou aux références (éléments, groupes…). La raison étant qu'une définition de contenu peut faire référence à des définitions d'un autre espace de noms. Pour que les références (via la valeur de l'attribut `ref`) soient trouvées sans ambiguïtés, il va falloir les qualifier et cela même si ces références se rapportent à des définitions du même schéma.

Un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.monentreprise.com/secretariat"
xmlns:s="http://www.monentreprise.com/secretariat">
<xs:element name="service">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="s:employe"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="employe">
 <xs:complexType>
  <xs:attribute name="nom" type="xs:string"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Ce schéma gère l'espace de noms `http://www.monentreprise.com/secretariat`. Les définitions globales `service` et `employe` sont donc d'office dans cet espace de noms. Comme l'élément `service` contient un élément `employe` par référence, nous sommes obligés de qualifier cette référence dans l'espace de noms du schéma par le préfixe `s`.

Autre exemple en passant par un type global :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.monentreprise.com/secretariat"
```

```
xmlns:s="http://www.monentreprise.com/secretariat">
<xs:element name="service" type="s:serviceType"/>

<xs:complexType name="serviceType">
<xs:sequence>
  <xs:element name="employe" form="qualified">
   <xs:complexType>
    <xs:attribute name="nom" type="xs:string"/>
   </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

Le type `serviceType` étant global (il possède un nom et se situe sous le schéma), il appartient à l'espace de noms géré par le schéma (`http://www.monentreprise.com/secretariat`). L'usage de ce type nécessite donc de lever l'ambiguïté de sa provenance par un préfixe (ici `s`).

La déclaration dans un document XML

Lorsque la racine de votre document se situe dans un espace de noms, il est obligatoire d'utiliser l'attribut `xsi:schemaLocation`. Celui-ci contient un ensemble de couples espace de noms et schéma. En règle générale, c'est plutôt le couple espace de noms de la racine et schéma lié qui est présent.

```
<service
xmlns="http://www.monentreprise.com/secretariat"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd">
<employe nom="dupond"/>
</service>
```

Dans cet exemple, le schéma `service.xsd` est lié à l'espace de noms `http://www.monentreprise.com/secretariat`.

La gestion des éléments locaux

Les éléments locaux (ils ne sont pas réutilisables par référence ou héritage) n'appartiennent pas d'office à l'espace de noms défini par l'attribut `targetNamespace`. C'est une particularité qu'il faut prendre en compte. Nous verrons, par la suite, qu'il est cependant possible d'établir une règle par défaut différente.

Prenons cet exemple nommé `service.xsd` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.monentreprise.com/secretariat">
<xs:element name="service">
 <xs:complexType>
 <xs:sequence>
```

```
    <xs:element name="employe">
     <xs:complexType>
      <xs:attribute name="nom" type="xs:string"/>
     </xs:complexType>
    </xs:element>
  </xs:sequence>
  </xs:complexType>
 </xs:element>
 </xs:schema>
```

Ici, l'élément `employe` est défini dans l'élément `service` ; il est donc local et n'a pas d'espace de noms, contrairement à `service` qui est global et appartient donc à l'espace de noms défini par l'attribut `targetNamespace`.

Voici deux exemples de documents XML qui sont valides par rapport au schéma `service.xsd` précédent :

```
<service
xmlns="http://www.monentreprise.com/secretariat"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd">
 <employe nom="dupond" xmlns=""/>
</service>

<s:service
xmlns:s="http://www.monentreprise.com/secretariat"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd">
 <employe nom="dupond"/>
</s:service>
```

Revenons à présent à la règle concernant les définitions locales. Cette règle par défaut, qui n'entraîne pas l'association avec l'espace de noms lié au schéma, n'est ni pratique ni réaliste car les espaces de noms s'appliquent généralement sur un contenu. Pour changer de règle on dispose des *switch* de qualification `elementFormDefault` et `attributeFormDefault`. Ces derniers se positionnent sur la racine du schéma et désignent le traitement des définitions locales pour les éléments et attributs. Ils prennent pour valeur `qualified` (ils appartiennent donc d'office à l'espace de noms lié au schéma) ou `unqualified`. En général, on considère qu'il est plus cohérent d'utiliser les valeurs `elementFormDefault="qualified"` et `attributeFormDefault="unqualified"`. Si nous avions utilisé ces valeurs dans le schéma précédent, un document XML valide pourrait donc être :

```
<s:service
xmlns:s="http://www.monentreprise.com/secretariat"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd">
<s:employe nom="dupond"/>
</s:service>
```

Il est également possible de mener une gestion plus fine des éléments ou attributs locaux par l'attribut `form`.

Par exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.monentreprise.com/secretariat"
xmlns:s="http://www.monentreprise.com/secretariat"
 elementFormDefault="unqualified"
 attributeFormDefault="qualified">
<xs:element name="service">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="employe" form="qualified">
    <xs:complexType>
     <xs:attribute name="nom" type="xs:string" form="unqualified"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Dans cet exemple, nous avons explicitement qualifié l'élément local `employe` et rendu non qualifié l'attribut `nom` (à l'inverse de la règle par défaut sur la racine du schéma).

Exercice 1

Espace de noms

Reprenez le document livre que nous avons réalisé dans le précédent chapitre et faites en sorte que tous les éléments soient dans l'espace de noms `http://www.masociete.com/livre`.

Conséquence de l'inclusion avec les espaces de noms

L'inclusion a un rôle passif pour la gestion des espaces de noms, c'est-à-dire que le schéma inclus épouse l'espace de noms du schéma l'incluant.

Supposons, par exemple, que nous disposions du schéma `service.xsd` suivant :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:s="http://www.monentreprise.com/secretariat" elementFormDefault="qualified">
<xs:element name="service" type="serviceType"/>
<xs:complexType name="serviceType">
  <xs:sequence>
   <xs:element name="employe">
    <xs:complexType>
     <xs:attribute name="nom" type="xs:string"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Ce schéma n'est pas lié à un espace de noms. Nous précisons cependant que les espaces de noms des élément locaux doivent toujours être qualifiés (en vue d'une inclusion avec un schéma lié à un espace de noms) par l'attribut `elementFormDefault`.

Voici un schéma effectuant l'inclusion :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.monentreprise.com/secretariat"
xmlns:s="http://www.monentreprise.com/secretariat"
 elementFormDefault="qualified">
 <xs:include schemaLocation="service.xsd"/>
 <xs:element name="secretariat" substitutionGroup="s:service"/>
</xs:schema>
```

L'élément `service` du schéma `service.xsd` devient lié à l'espace de noms `http://www.monentreprise.com/secretariat`.

Utilisation de l'importation pour les espaces de noms

L'importation est un mécanisme proche de l'inclusion mais concerne les espaces de noms. L'importation désigne un espace de noms et un ensemble de définitions en provenance d'un schéma. Il est possible de ne pas préciser la localisation du schéma qui correspond à un espace de noms : la résolution entre l'espace de noms et l'URI du schéma se fait alors dans le document XML (via l'attribut `schemaLocation`).

L'importation s'effectue sous la racine du schéma tout d'abord grâce à l'élément `import`, puis par les attributs `namespace` et `schemaLocation`.

Soit le schéma suivant, `employe.xsd`, lié à l'espace de noms `http://www.employe.com` :

```
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.employe.com">
<xs:element name="employe">
 <xs:complexType>
  <xs:attribute name="nom" type="xs:string"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

L'importation dans le schéma `service.xsd` lié à l'espace de noms `http://www.monentreprise.com/secretariat` peut se faire ainsi :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.monentreprise.com/secretariat"
    xmlns:s="http://www.monentreprise.com/secretariat"
    xmlns:e="http://www.employe.com"
    elementFormDefault="qualified">
<xs:import namespace="http://www.employe.com" schemaLocation="employe.xsd"/>
<xs:element name="service" type="s:serviceType"/>
<xs:complexType name="serviceType">
 <xs:sequence>
```

```
  <xs:element ref="e:employe"/>
 </xs:sequence>
</xs:complexType>
</xs:schema>
```

Un document XML valide serait celui-ci :

```
<s:service
xmlns:s="http://www.monentreprise.com/secretariat"
xmlns:e="http://www.employe.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd">
 <e:employe nom="dupond"/>
</s:service>
```

Nous avons donc bien validé un document imbriquant plusieurs espaces de noms.

Vous remarquerez que nous avons défini la relation entre les schémas `service.xsd` et `employe.xsd` de manière explicite. Il est également possible de ne rendre implicite cette liaison qu'à travers le document XML en écrivant pour le schéma `service.xsd` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.monentreprise.com/secretariat"
    xmlns:s="http://www.monentreprise.com/secretariat"
    xmlns:e="http://www.employe.com"
    elementFormDefault="qualified">
<xs:import namespace="http://www.employe.com"/>
<xs:element name="service" type="s:serviceType"/>
<xs:complexType name="serviceType">
 <xs:sequence>
  <xs:element ref="e:employe"/>
 </xs:sequence>
</xs:complexType>
</xs:schema>
```

Comme vous pouvez le remarquer, l'attribut `schemaLocation` a disparu. À ce stade, le parseur ne sait donc pas où se trouve la définition de l'élément `employe`. Pour la lui signaler, nous écrivons dans le document XML ceci :

```
<s:service
xmlns:s="http://www.monentreprise.com/secretariat"
xmlns:e="http://www.employe.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.monentreprise.com/secretariat service.xsd
➥http://www.employe.com employe.xsd">
 <e:employe nom="dupond"/>
</s:service>
```

L'attribut `schemaLocation` résout donc aussi la localisation des schémas importés. Cela offre une certaine flexibilité, par exemple dans le cas de documents de versions différentes.

Exercice 2

Importation

Soit le document `ab.xml` suivant :

```
<a:A xmlns:a="http://www.a.com" xmlns:b="http://www.b.com">
 <b:B>un texte
 </b:B>
</a:A>
```

L'objectif est de valider ce document à l'aide des schémas `a.xsd` et `b.xsd` respectivement liés à l'espace de noms `http://www.a.com` et `http://www.b.com`.

Exercice 3

Importation et éléments locaux

```
<a:A xmlns:a="http://www.a.com" xmlns:b="http://www.b.com">
<b:B>
    <b:C>
    Un texte
    </b:C>
</b:B>
</a:A>
```

1. Créez une nouvelle version `a2.xsd` et `b2.xsd` à partir des schémas `a.xsd` et `b.xsd`. Validez ce document.

2. Modifiez à nouveau `b2.xsd` pour cacher l'espace de noms de l'élément `C`.

Quelle conclusion pouvez-vous en tirer ?

Parallèle avec la conception objet

Comme nous allons le voir, il existe des analogies entre le monde objet et les schémas XML. Ces similitudes peuvent donner des moyens supplémentaires pour représenter des structures, par exemple, en s'appuyant sur UML (*Unified Modeling Language*).

Quelques rappels de programmation objet

Quelques rappels sur les concepts objet sont indispensables pour comprendre cette partie.

Tout d'abord, en programmation objet, une classe est un squelette qui sert à produire des objets. Ce squelette définit les états possibles de chaque objet sous la forme d'attributs

(variables dont la portée est la classe). Il définit également des actions qui donnent vie à l'objet. Les actions n'ayant pas vraiment de sens au niveau XML, nous allons les ignorer.

Les classes peuvent être réutilisées pour créer d'autres classes par dérivation. En programmation objet, on parle d'extension d'une classe, c'est-à-dire qu'une classe dérivée est toujours un sur-ensemble, en termes de capacité, de la classe parent.

La dérivation peut généralement être contrôlée par la notion de classe finale (pas de classes dérivées autorisées).

Un objet est une instance d'une classe au sens où il est construit via le squelette de la classe et peut ensuite suivre ses propres évolutions d'état en fonction des actions que l'on réalise.

Parfois, on souhaite contrôler la création d'objets. On rend alors certaines classes abstraites, ce qui interdit toute création d'objets. Les classes abstraites sont souvent liées à un concept de *framework* : elles servent de couches de base et contiennent le code et les attributs les plus communs ; certains traitements sont ensuite délégués aux classes dérivées davantage liées au système sur lequel est effectué le développement.

L'emboîtement des classes parent et enfant rend l'objet multiple. Il s'appuie à la fois sur le squelette de la classe qui lui a donné naissance mais aussi sur ceux des parents qui ont donné naissance à sa classe. Cette particularité simplifie parfois la gestion de groupe d'objets en se focalisant sur des classes communes. Par exemple, si une classe `figure` représente une figure géométrique dans un espace euclidien, toutes les formes de figure peuvent dériver de cette classe. Mais, si nous avons besoin de déplacer un ensemble de figures, la nature de la figure importe peu : seule leur parenté avec la classe `figure` compte. Ce concept, qui s'attache à associer un objet à une classe ancêtre, s'appelle le polymorphisme. Contrairement à ce que laisserait entendre ce terme, ce n'est pas l'objet qui change d'aspect, c'est simplement la manière dont on le manipule.

Lorsqu'il y a dérivation, il peut y avoir une réécriture de certaines actions. Ce concept s'appelle la surcharge de classes.

Lien entre type et classe

Dans les schémas, les types sont similaires à des classes, qu'ils soient simples ou complexes. La dérivation des types fonctionne par extension ou restriction (ce dernier cas n'ayant pas de sens en programmation objet).

Voici un exemple d'une dérivation d'un type générique `personneType` en `employeType`.

```
<xs:complexType name="personneType">
 <xs:sequence>
  <xs:element name="nom" type="xs:string"/>
  <xs:element name="prenom" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="employeType">
 <xs:complexContent>
```

```
  <xs:extension base="personneType">
   <xs:attribute name="id" type="xs:token"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Lien entre l'élément et l'objet

L'objet et l'élément sont très proches. La structure d'un élément, qu'elle soit interne ou externe, suit bien le rapprochement avec la classe.

Dans l'exemple suivant :

```
<xs:element name="personne" type="personneType"/>
```

parler d'objet `personne` ou d'instance `personne` est analogue.

Dans cet autre exemple :

```
<xs:element name="employe" type="employeType"/>
<xs:element name="boss" type="employeType"/>
```

les éléments `employe` et `boss` partagent une même structure et donc un même type.

Lien entre la substitution d'élément et le polymorphisme

La substitution se rapproche du principe du polymorphisme. On désigne un élément initial comme porteur d'une définition (son contenu). Un substitut peut exister s'il a la même définition ou si sa définition est compatible, par dérivation, avec celle de l'élément initial.

Un substitut est désigné par l'attribut `substitutionGroup`. Cela n'a de sens que pour les éléments définis globalement.

Voici une partie d'un schéma :

```
<xs:element name="employe" type="personneType"/>
<xs:element name="entreprise">
<xs:complexType>
 <xs:sequence>
  <xs:element ref="employe" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="boss" substitutionGroup="employe"/>
```

Comme nous pouvons le voir, l'élément `entreprise` contient une séquence d'éléments `employe`. Comme l'élément `boss` est un substitut de l'élément `employe`, l'élément `entreprise` contient donc en réalité une séquence d'éléments `employe` ou `boss`. Pour résumer, partout où l'élément `employe` est présent, l'élément `boss` peut également être utilisé.

Voici un exemple de document XML valide pour ce schéma :

```
<entreprise xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="employe.xsd">
<employe>
 <nom>Doe</nom>
 <prenom>John</prenom>
 <telephone>999</telephone>
</employe>
<boss>
 <nom>Durand</nom>
 <prenom>Yves</prenom>
 <telephone>888</telephone>
</boss>
</entreprise>
```

Voici un exemple plus sophistiqué vous montrant comment réaliser un substitut ayant un type dérivé.

```
<xs:element name="service" substitutionGroup="entreprise">
<xs:complexType>
 <xs:complexContent>
  <xs:restriction base="entrepriseType">
   <xs:sequence>
    <xs:element ref="employe" maxOccurs="10"/>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
</xs:element>

<xs:complexType name="entrepriseType">
 <xs:sequence>
  <xs:element ref="employe" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
<xs:element name="entreprise" type="entrepriseType"/>
```

L'élément `entreprise` ne peut contenir qu'une séquence d'éléments `employe`. Nous avons cependant réalisé un substitut `service` qui limite le nombre d'éléments `employe` à dix, grâce à une dérivation par restriction.

Exercice 4

Substitution

Soit le document `employe.xml` :

```
<groupe>
<personne nom="Dupond" prenom="Jacques"/>
<employe nom="Durand" prenom="Jules">
 <fonction>
```

```
Formateur
 </fonction>
</employe>
<personne nom="Dupont" prenom="Jean"/>
...
</groupe>
```

Le groupe contient un ensemble d'éléments compatibles avec l'élément `personne` par substitution.

Créez le schéma `employe.xsd` et validez ce document.

Lien entre l'abstraction d'élément et la classe abstraite

La classe abstraite interdit toute création d'objets. De manière similaire, un élément abstrait ne peut être employé dans le document XML. Il ne sert qu'au schéma et n'a de sens que s'il y a des substituts compatibles non abstraits.

Voici un exemple qui vous montre l'usage de l'attribut `abstract`.

```
<xs:element name="personne" type="personneType" abstract="true"/>
<xs:element name="employe" substitutionGroup="personne"/>
<xs:element name="boss" substitutionGroup="employe"/>
<xs:complexType name="entrepriseType">
 <xs:sequence>
  <xs:element ref="personne" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

Ici, l'élément `personne` est abstrait et ne peut donc être directement employé dans un document XML. Seuls les substituts `employe` ou `boss` pourront être utilisés.

Lien entres les différentes formes de contrôle et les limitations de dérivation de classe

Il est possible d'empêcher que certains éléments soient substitués grâce à l'attribut `block`.

Exemple :

```
<xs:element name="personne" type="personneType" abstract="false"  block
➥="substitution"/>
<xs:element name="employe" substitutionGroup="personne"/>
<xs:element name="boss" substitutionGroup="employe"/>
```

Ici, la substitution a été interdite pour l'élément `personne`. Il n'est donc pas abstrait pour être employable.

Le contrôle peut aussi s'appliquer à la dérivation des types par restriction ou extension grâce à l'attribut `final`. Cet attribut peut prendre les valeurs `restriction`, `extension` ou `#all` pour définir le type de blocage. Cette notion se rapproche des classes finales.

Exemple :

```
<xs:complexType name="entrepriseType" final="extension">
<xs:sequence>
  <xs:element ref="personne" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

Dans cet exemple, le type complexe `entrepriseType` ne peut pas servir à la création d'un nouveau type par extension, mais la réalisation d'un autre type par restriction reste possible (en jouant sur les cardinalités, par exemple).

Lien entre la surcharge d'un type et la surcharge de méthode

Dans le document XML (que l'on peut aussi qualifier d'instance du schéma), il est possible de changer le type d'un élément par un autre dérivé de ce dernier. Cela est possible via l'attribut `xsi:type`.

Exemple :

```
<xs:complexType name="employeType">
 <xs:complexContent>
  <xs:extension base="personneType">
   <xs:sequence>
    <xs:element name="contrat" type="xs:token"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

Cet extrait d'un schéma suit celui que nous avons utilisé avec `employe` et `boss` comme substituts de l'élément `personne`. Nous y avons déclaré un type `employeType` qui dérive par extension du type `personneType` en y ajoutant un élément `contrat`.

Voici comment nous allons pouvoir utiliser ce type dans un document XML :

```
<entreprise xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="employe.xsd">
<boss>
 <nom>Durand</nom>
 <prenom>Yves</prenom>
</boss>
<employe xsi:type="employeType">
 <nom>Durand</nom>
 <prenom>Yves</prenom>
 <contrat>CDD</contrat>
</employe>
</entreprise>
```

Comme vous pouvez le voir, notre élément `employe`, qui n'était qu'un substitut sans modification de type de l'élément `personne`, peut maintenant contenir un élément `contrat`.

Ce système de changement de type dans le document XML peut être contrôlé grâce à l'attribut `block` sur un élément. Les valeurs `extension`, `restriction` et `#all` servent à définir le type de limitation.

Un exemple :

```
<xs:element name="personne" type="personneType" abstract="false"/>
<xs:element name="employe" substitutionGroup="personne" block="extension"/>
```

Cet extrait rend impossible l'usage d'un autre type dérivé par extension sur l'élément `employe`. Le cas précédent avec `employeType` ne serait donc plus possible.

Exercice 5

Substitution de type

Soit le document `employe2.xml` suivant :

```
<groupe>
<personne nom="Dupond" prenom="Jacques"/>
<personne nom="Durand" prenom="Jules"/>
<fonction>
Formateur
</fonction>
</personne>
...
</groupe>
```

Validez ce document à l'aide du schéma précédent (`employe.xsd`). Ajoutez et testez un type supplémentaire `directeurType` (en passant par un autre schéma `employe2.xsd`) avec comme contenu un élément `entreprise` contenant lui-même le nom d'une société.

Cas des éléments vides

Lorsqu'on réalise un document XML, il est certain que certaines parties ne pourront pas être renseignées. On peut imaginer que certaines données ne soient pas encore collectées ou toutes disponibles en même temps. Néanmoins, pour valider le document, nous disposons de l'attribut `xsi:nil` qui prend la valeur `true` si l'élément qui le contient n'a pas encore de contenu et doit donc être ignoré par le parseur. Pour que cet attribut puisse être utilisé, il faut ajouter dans le schéma l'attribut `nillable` sur la définition de l'élément.

Un exemple :

```
<xs:element name="personne" type="personneType"  nillable="true"/>
<xs:element name="employe" substitutionGroup="personne" nillable="true"/>
<xs:element name="boss" substitutionGroup="employe"/>
```

Dans cet extrait de schéma, nous autorisons la présence des éléments `personne` et `employe` sans contenu.

Voici un extrait d'un document XML valide correspondant :

```
<boss>
 <nom>Durand</nom>
 <prenom>Yves</prenom>
</boss>
<employe xsi:nil="true"/>
<personne xsi:nil="true"/>
<boss xsi:nil="true"/>
```

Patrons (Design patterns)

Le concept de design pattern consiste à formuler une méthode de fabrication. Ces patrons de conception fournissent donc un cadre de construction pour faciliter certains usages. Ils sont courants en programmation objet pour la construction, l'organisation et le comportement des objets. Les design patterns ne sont cependant pas une solution miracle à tous les problèmes : ils ont leurs avantages et leurs inconvénients.

Nous examinons ci-après quelques patrons classiques.

Design pattern : les poupées russes

Ce patron peut être vu comme un ensemble de poupées russes, qui s'emboîtent les unes dans les autres. Cet emboîtement achevé, seule la poupée la plus grosse est visible. Ce principe est mis en œuvre lorsqu'un schéma s'appuie sur des définitions locales et/ou des types anonymes (sans nom). Les définitions s'emboîtent alors les unes dans les autres.

Voici un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="personne">
<xs:complexType>
<xs:sequence>
  <xs:element name="adresse">
   <xs:complexType>
   ...
   </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Points négatifs : peu de visibilité et pas de réutilisation.

Points positifs : contrôle de la qualification par le switch `elementFormDefault` ; un élément de même nom peut posséder plusieurs définitions (notion de contexte d'usage).

Design pattern : les tranches de salami

Dans ce design, chaque élément est défini globalement ; les interconnexions de contenus (séquence, choix…) sont réalisées par référence. Chaque élément peut être vu comme une tranche de salami : on les positionne les uns à côté des autres.

Voici un exemple de ce design :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="personne">
<xs:complexType>
  <xs:sequence>
   <xs:element ref="adresse"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="adresse">
 <xs:complexType>
 ...
 </xs:complexType>
</xs:element>
</xs:schema>
```

Les éléments `personne` et `adresse` sont bien définis globalement et l'élément `personne` contient, dans sa définition, une référence à l'élément `adresse`.

Points négatifs :

- Pas de typage réemployable (types anonymes).
- Pas de dérivation.
- Pas de contexte d'usage d'un élément (tous globaux).

Points positifs :

- Substitution possible.
- Cohérence avec les espaces de noms.
- Réutilisation d'élément par référence ou dans un autre schéma par inclusion ou importation.

Design pattern : les stores vénitiens

Un store vénitien est composé d'un ensemble de planches reliées par deux cordes. Dans ce design, les types sont tous globaux (jouant le rôle de la planche).

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="personne" type="personneType"/>
<xs:complexType name="personneType">
 <xs:sequence>
  <xs:element name="adresse" type="adresseType"/>
 </xs:sequence>
```

```
</xs:complexType>
<xs:complexType name="adresseType">
...
</xs:complexType>
</xs:schema>
```

Point négatif : pas de substitution, hormis pour la racine.

Points positifs :

- Réutilisation de type.
- Dérivation de type.
- Facilite la création de librairies.
- Contrôle de la qualification par le switch `elementFormDefault`.
- Un élément de même nom peut posséder plusieurs définitions (contexte d'usage via le type).

Design pattern : la forme mixte

Le design mixte mêle les patrons tranche de salami et stores vénitiens. L'idée est de bénéficier à la fois d'une librairie d'éléments, mais également de types. Les définitions globales sont donc favorisées au détriment des définitions locales.

Voici un exemple de schéma :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="personne" type="personneType"/>
<xs:element name="adresse" type="adresseType"/>
<xs:complexType name="personneType">
 <xs:sequence>
  <xs:element ref="adresse"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="adresseType">
...
</xs:complexType>
</xs:schema>
```

On voit bien que les éléments `personne` et `adresse` sont employables par référence ou par substitution. Les types `personneType` et `adresseType` sont utilisables pour des contenus d'éléments, mais également par dérivation.

Points négatifs :

- Pas de contrôle de la qualification par le switch `elementFormDefault`.
- Pas de répétition d'un nom d'élément.

Points positifs :

- Substitution des éléments.
- Réutilisation de type.
- Dérivation de type.
- Facilite la création de librairies.

C'est en général ce type de design, dit ouvert, qu'il vaut mieux favoriser, quitte à utiliser certains switchs de restriction, comme `block`, `abtract` ou `final`.

Modélisation avec héritage ou avec groupe

L'un des problèmes de la composition objet est qu'il existe deux stratégies pour agglomérer des classes afin d'optimiser l'architecture objet. La première stratégie est la délégation : une classe possède un attribut (pseudo-variable globale dont la portée est la classe) et un objet, qui fournit un service. La deuxième stratégie, la dérivation, permet d'obtenir un service en définissant une relation de parenté. On le comprendra bien, cette dernière solution n'aura de sens que si la classe fille est de même forme que la classe parent. Dans le cas contraire, les risques de créer une architecture artificielle trop concentrée sur la réalisation d'un service et peu flexible sont importants.

Le problème du choix entre héritage ou délégation se pose également dans la réalisation d'un schéma. On peut effectuer une dérivation d'un type par extension ou restriction, ou bien associer des groupes de définition (donc une délégation).

La modélisation avec héritage

Nous l'avons vu dans le chapitre précédent : l'héritage s'applique aux types simples et complexes. Il nécessite d'avoir une vision très précise des points stables dans une structure XML.

Un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="avecIdType">
 <xs:sequence>
  <xs:element name="id" type="xs:token"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="personneType">
 <xs:complexContent>
  <xs:extension base="avecIdType">
   <xs:sequence>...</xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Dans ce schéma, nous avons créé un type `avecIdType` utilisable par dérivation (cas avec `personneType`).

Points négatifs :

- Impact des changements significatifs au sommet de la hiérachie.
- Nécessité de restreindre les dérivations sur certains types.

Points positifs :

- Proche de l'objet.
- Relation entre les types.
- Hiérarchie documentable de type.
- Affinage des types (spécialisation par la dérivation).
- Substitution avec dérivation.
- Altération de type dans un document (attribut `xsi:type`).

La modélisation avec groupe

Les groupes agglomèrent des blocs de définitions. Il n'y a pas de relations claires entre les groupes (principal, secondaire…). Ils sont d'un usage intéressant lorsque des éléments indépendants (sémantiquement parlant) ont des points de structure en commun. On peut faire une analogie avec l'héritage multiple qui tente de faire cohabiter plusieurs parents pour une classe.

Un exemple de schéma :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:group name="avecId">
 <xs:sequence>
  <xs:element name="id" type="xs:token"/>
 </xs:sequence>
</xs:group>
<xs:complexType name="personneType">
 <xs:complexContent>
  <xs:sequence>
   <xs:group ref="avecId"/>
  ...
  </xs:sequence>
 </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Dans cet exemple, nous avons essayé de reproduire le cas que nous avons abordé par héritage avec un identifiant. Le résultat semble ici similaire. Cependant, il faut bien souligner que le groupe `avecId` peut être employé aussi dans des types totalement indépendants de `personneType`.

Points négatifs :

- Manque de souplesse sur les évolutions (difficile d'estimer l'impact d'un changement dans un groupe car pas de vision sémantique des usages).
- Documentation plus délicate car pas de hiérarchie (qui utilise quoi ?).
- Pas de restriction possible (on ne peut réduire un groupe).

Points positifs :

- Allège la conception des schémas en évitant des dérivations inutiles et émule l'héritage multiple.
- Concentre les parties communes de définitions sans rapport (notion de factorisation).

La modélisation avec groupe et héritage

Les dérivations et les groupes peuvent cohabiter à des niveaux différents, les groupes sont plutôt utilisés pour réaliser des types génériques. La portée de ces groupes s'en trouve ainsi réduite et évite un éparpillement difficilement contrôlable.

Voici un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:group name="avecId">
 <xs:sequence>
  <xs:element name="id" type="xs:token"/>
 </xs:sequence>
</xs:group>
<xs:complexType name="communType" abstract="true">
<xs:sequence>
 <xs:group ref="avecId"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="personneType">
<xs:complexContent>
 <xs:extension base="communType">
   <xs:sequence>
    ...
  </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
</xs:schema>
```

On voit bien que le groupe `avecId` n'a servi qu'aux types de base `communType`. Le type `personneType` est alors dérivé de `communType`.

Points négatifs :

- Complexité.
- Travail supplémentaire (préparation de groupes).

Points positifs :

- Maximum de souplesse horizontale et verticale.
- La composition par groupes est limitée aux types abstraits (impact limité).

Modélisation avec les espaces de noms

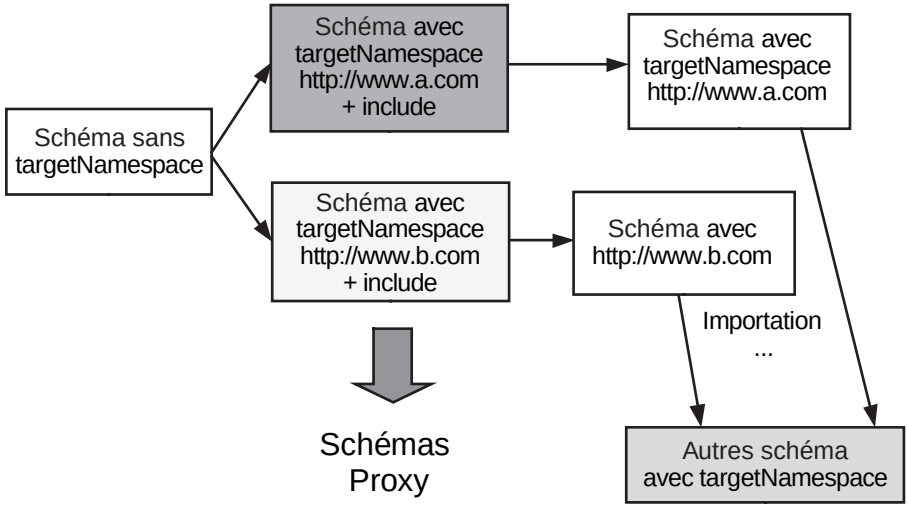
Malgré leur côté complexe, il est possible de réaliser des structures indépendantes des espaces de noms. Elles pourront prendre plusieurs « identités ».

Modélisation par le design caméléon

Dans un design dit caméléon, l'objectif est de s'appuyer sur l'élément `include` qui a la particularité d'emprunter le `targetNamespace` du schéma utilisateur.

L'idée est de construire un schéma sans `targetNamespace` (le caméléon) et de l'inclure dans un schéma intermédiaire, que l'on peut appeler schéma *proxy*, mais qui, lui, utilise un `targetNamespace`. Cela a pour effet de créer plusieurs versions de notre premier schéma applicables à des espaces de noms distincts.

Figure 4–1
Exemple de schémas proxy



Dans la figure 4-1, le schéma le plus à gauche joue le rôle de caméléon. En effet, nous obtenons deux schémas proxy pour les espaces de noms `http://www.a.com` et `http://www.b.com`, ces schémas étant ensuite utilisés par importation dans d'autres schémas.

Modélisons maintenant un autre cas.

Soit `cam1.xsd`, le schéma caméléon :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="personne" type="xs:string"/>
</xs:schema>
```

Et `cam2.xsd`, le schéma proxy :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  targetNamespace=
➥"http://www.secretariat.com">
<xs:include schemaLocation="cam1.xsd"/>
</xs:schema>
```

Enfin, `cam3.xsd` est le schéma utilisateur :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
➥"http://www.service.com"
xmlns:p="http://www.secretariat.com">
<xs:import namespace="http://www.secretariat.com" schemaLocation="cam2.xsd"/>
<xs:element name="service">
 <xs:complexType>
  <xs:sequence maxOccurs="unbounded">
   <xs:element ref="p:personne"></xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Un document XML valide pourrait être :

```
<service xmlns="http://www.service.com"   xmlns:s="http://www.secretariat.com"...>
<s:personne></s:personne>
</service>
```

Le passage par le proxy `cam2.xsd` a permis l'association entre l'élément `personne` et l'espace de noms `http://www.secretariat.com`.

Exercice 6

Utilisation d'un design caméléon

Créez un document `ab3.xml` :

```
<A
xmlns="http://www.a.com"
xmlns:b="http://www.b.com"
xmlns:c="http://www.c.com">
<b:B>Un texte</b:B>
<c:B>10</c:B>
</A>
```

Créez les schémas `b3.xsd` (premier élément `B`) et `c3.xsd` (deuxième élément `B`) sans espace de noms. Créez les proxys `b3-proxy.xsd` et `c3-proxy.xsd` pour les espaces de noms respectifs `http://www.b.com` et `http://www.c.com`.

Réalisez un schéma `a3.xsd` utilisant `b3-proxy.xsd` et `c3-proxy.xsd`. Validez le document `ab3.xml`.

Quelle conclusion pouvez-vous en tirer ?

Les définitions neutres dans un schéma

Lorsqu'on réalise un schéma, il arrive que la globalité des utilisations ne soit pas connue, pour la simple raison que la modélisation d'un problème suit plusieurs évolutions et que les structurations des données évoluent en conséquence. L'idée est alors d'ouvrir le schéma en filtrant des éléments, par exemple, selon un espace de noms.

Utilisation de any

Les instructions `any` ou `anyAttribute` permettent d'ouvrir un schéma à des parties non prévues initialement. Pour le cas de l'élément, l'attribut `processContents` va indiquer au parseur comment il doit réagir lorsqu'il rencontre des éléments n'appartenant pas au schéma.

Voici un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="personne">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="nom" type="xs:string"/>
    <xs:element name="prenom" type="xs:string"/>
    <xs:any processContents="skip" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Dans ce schéma, l'élément `personne` contient une séquence de `nom` et `prenom` et d'autres éléments en nombre indéterminé et d'origine inconnue.

L'attribut `processContents` peut prendre les valeurs `skip` (pas de validation), `strict` (validation effectuée) et `lax` (recherche d'un schéma pour validation mais pas de notification d'erreur si ce schéma n'est pas trouvé).

Voici un document XML valide, cohérent au schéma ci-avant :

```
<personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="ouverture1.xsd">
 <nom>Mr dupont</nom>
```

```
 <prenom>Jean</prenom>
 <adresse>La défense</adresse>
 <email>j.dupont@hotmail.com</email>
</personne>
```

L'attribut `namespace` peut également être associé à `processContents` pour définir les espaces de noms acceptés.

Voici un exemple :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.e.com"
 elementFormDefault="qualified">
<xs:element name="personne">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="nom" type="xs:string"/>
   <xs:element name="prenom" type="xs:string"/>
    <xs:any processContents="skip" maxOccurs="unbounded" namespace=
    ➥"http://www.f.com"></xs:any>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Ici, nous avons seulement autorisé la présence des éléments dont l'espace de noms est `http://www.f.com`.

Un document XML valide pourrait être :

```
<personne xmlns="http://www.e.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.e.com ouverture1.xsd">
<nom>Mr dupont</nom>
<prenom>Jean</prenom>
<adresse xmlns="http://www.f.com">La défense</adresse>
</personne>
```

À noter que l'attribut `namespace` peut prendre les valeurs suivantes :

- `##local` : l'élément est alors non qualifié.
- `##targetNamespace` : l'élément est dans le même espace de noms que le `targetNamespace` du schéma.
- `##any` : l'élément peut être dans n'importe quel espace de noms.
- `##other` : l'élément est dans un autre espace de noms que le `targetNamespace` du schéma.
- Liste d'URI : on insère plusieurs espaces de noms séparés par un espace.

Correction des exercices

L'ensemble des exercices a été réalisé sur le logiciel EditiX (*http://www.editix.com/*). Une version d'évaluation de 30 jours est librement téléchargeable (*http://www.editix.com/download.html*).

Exercice 1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema targetNamespace="http://www.masociete.com/livre" xmlns:xs=
➥"http://www.w3.org/2001/XMLSchema" elementFormDefault=
➥"qualified" attributeFormDefault="unqualified"
➥xmlns:soc="http://www.masociete.com/livre">
<xs:complexType name="livreType">
 <xs:sequence>
  <xs:element name="auteurs" type="soc:auteursType"/>
  <xs:element name="sections" type="soc:sectionsType"/>
 </xs:sequence>
 <xs:attributeGroup ref="soc:avecTitre"/>
</xs:complexType>
<xs:complexType name="auteursType">
 <xs:group ref="soc:auteursGrp"/>
</xs:complexType>
<xs:group name="auteursGrp">
 <xs:sequence>
  <xs:element name="auteur" type="soc:auteurType" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:group>
<xs:attributeGroup name="avecTitre">
 <xs:attribute name="titre" type="xs:string" use="required"/>
</xs:attributeGroup>
<xs:complexType name="auteurType">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="nom" type="xs:string" use="required"/>
   <xs:attribute name="prenom" type="xs:string" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
<xs:complexType name="sectionsType">
 <xs:sequence>
  <xs:element name="section" type="soc:sectionType" minOccurs="2" maxOccurs=
  ➥"unbounded"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="sectionType">
 <xs:sequence>
  <xs:group ref="soc:auteursGrp" minOccurs="0"/>
  <xs:element name="chapitre" type="soc:chapitreType" minOccurs="2" maxOccurs=
  ➥"unbounded"/>
```

```
 </xs:sequence>
 <xs:attributeGroup ref="soc:avecTitre"/>
</xs:complexType>
<xs:complexType name="chapitreType">
 <xs:sequence>
  <xs:element name="paragraphe" type="xs:string" minOccurs="2" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attributeGroup ref="soc:avecTitre"/>
</xs:complexType>
<xs:element name="livre" type="soc:livreType"/>
</xs:schema>
```

Exercice 2

b.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault=
➥"qualified" attributeFormDefault="unqualified" targetNamespace=
➥"http://www.b.com">
<xs:element name="B" type="xs:string"/>
</xs:schema>
```

a.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="
➥qualified" attributeFormDefault="unqualified" targetNamespace=
➥"http://www.a.com" xmlns:a="http://www.a.com" xmlns:b="http://www.b.com">
<xs:import namespace="http://www.b.com" schemaLocation="b.xsd"/>
<xs:complexType name="AType">
 <xs:sequence>
  <xs:element ref="b:B"/>
 </xs:sequence>
</xs:complexType>
<xs:element name="A"  type="a:AType"/>
</xs:schema>
```

Le schéma `a.xsd` importe le schéma `b.xsd`.

ab.xml :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<a:A xmlns:a="http://www.a.com" xmlns:b="http://www.b.com" xmlns:xsi=
➥"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
➥"http://www.a.com a.xsd">
<b:B>
un texte
</b:B>
</a:A>
```

Exercice 3

b2.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
➥http://www.b.com" elementFormDefault="unqualified" attributeFormDefault=
➥"unqualified">
<xs:element name="B">
<xs:complexType>
 <xs:sequence>
   <xs:element name="C" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Remarquez bien la valeur de l'attribut `elementFormDefault`.

a2.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:a=
➥"http://www.a.com" xmlns:b="http://www.b.com" targetNamespace=
➥"http://www.a.com" elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:import namespace="http://www.b.com" schemaLocation="b2.xsd"/>
<xs:complexType name="AType">
<xs:sequence>
  <xs:element ref="b:B"/>
 </xs:sequence>
</xs:complexType>
<xs:element name="A" type="a:AType"/>
</xs:schema>
```

ab2.xml :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<a:A xmlns:a="http://www.a.com" xmlns:b="http://www.b.com" xmlns:xsi=
➥"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
➥"http://www.a.com a2.xsd">
 <b:B>
 <C>Un texte</C>
 </b:B>
</a:A>
```

Nous n'avons plus besoin de spécifier l'espace de noms de l'élément `C`.

Exercice 4

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="personneType">
<xs:attribute name="nom" type="xs:string" use="required"/>
 <xs:attribute name="prenom" type="xs:string" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="employeType">
<xs:complexContent>
  <xs:extension base="personneType">
   <xs:sequence>
   <xs:element name="fonction" type="xs:string"/>
  </xs:sequence>
 </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="groupeType">
<xs:sequence>
  <xs:element ref="personne" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
<xs:element name="groupe" type="groupeType"/>
<xs:element name="personne" type="personneType"/>
<xs:element name="employe" type="employeType" substitutionGroup="personne"/>
</xs:schema>
```

Exercice 5

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault=
➥"qualified" attributeFormDefault="unqualified">
<xs:include schemaLocation="employe.xsd"/>
<xs:complexType name="directeurType">
<xs:complexContent>
  <xs:extension base="personneType">
   <xs:sequence>
    <xs:element name="entreprise" type="xs:string"/>
  </xs:sequence>
   </xs:extension>
 </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Notre schéma `employe2.xsd` inclut le schéma `employe.xsd`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<groupe xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xsi:noNamespaceSchemaLocation="employe2.xsd">
<personne nom="Dupond" prenom="Jacques" xsi:type="personneType"/>
<personne nom="Durand" prenom="Jules" xsi:type="employeType">
<fonction>
 Formateur
</fonction>
</personne>
<personne nom="Durand" prenom="Jules" xsi:type="directeurType">
<entreprise>Toto</entreprise>
</personne>
</groupe>
```

Le dernier élément `personne` contient bien l'élément `entreprise`.

Exercice 6

b3.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="B" type="xs:string"/>
</xs:schema>
```

c3.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="B" type="xs:int"/>
</xs:schema>
```

b3-proxy.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
➥"http://www.b.com" elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:include schemaLocation="b3.xsd"/>
</xs:schema>
```

c3-proxy.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
➥"http://www.c.com">
<xs:include schemaLocation="c3.xsd"/>
</xs:schema>
```

a3.xsd :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace=
➥"http://www.a.com"  elementFormDefault="qualified" attributeFormDefault=
➥"unqualified" xmlns:b="http://www.b.com" xmlns:c="http://www.c.com">
<xs:import namespace="http://www.b.com" schemaLocation="b3-proxy.xsd"/>
<xs:import namespace="http://www.c.com" schemaLocation="c3-proxy.xsd"/>
<xs:element name="A" id="a1">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="b:B"/>
   <xs:element ref="c:B"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

ab3.xml :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<A xmlns="http://www.a.com" xmlns:b="http://www.b.com" xmlns:c=
➥"http://www.c.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
➥instance" xsi:schemaLocation="http://www.a.com
a3.xsd">
<b:B>Un texte</b:B>
<c:B>10</c:B>
</A>
```

5

Publication de documents XML

Ce chapitre traite de la publication de documents XML, qui passe essentiellement par le format final XHTML/HTML. Nous allons donc procéder à quelques rappels sur ce format mais également sur le format CSS (Cascading Style Sheets). Le traitement d'un document XML, en vue de sa publication, nécessite l'apprentissage du langage de requêtage XPath ainsi que de l'ensemble XSL (eXtensible Stylesheet Language).

Rôle de la publication

XML est un format universel et son rôle dans la représentation graphique est important. C'est un peu comme avec la programmation objet : XML est idéal pour représenter des arbres d'objets graphiques (hiérarchie indéfinie dans les deux cas).

Publication des données textes

L'intranet/Internet a pris une place importante dans les entreprises. Les navigateurs remplacent de plus en plus les applications riches et donnent un point d'accès unique. Comme nous l'avons vu, le format XML se charge d'agglomérer des données, ce qui facilite leur stockage ainsi que leur circulation. On s'attend donc nécessairement à ce que ces données soient visibles à un moment ou à un autre pour les utilisateurs, que ce soit à titre de contrôle, pour les personnels de l'entreprise, ou bien pour des clients (cadre B2C : *Business to consumer*). Cette visibilité doit éliminer tout sens technique. Un document XML, même structuré simplement, doit être présentable, c'est-à-dire être qu'il doit

être lié à une charte graphique (couleurs, polices de caractères…), contenir des tableaux, des images… En un mot, tout un ensemble qui n'existe pas dans le format initial. Les langages qui autorisent cela restent essentiellement ceux du Web, soit XHTML (*Extensible HyperText Markup Language*)/HTML et CSS. Mais il existe aussi d'autres formats employés dans l'entreprise comme PDF (*Portable Document Format*) ou RTF (*Rich Text Format*)… Par exemple, nous pourrions imaginer que des fiches produit décrites en XML soient disponibles à la fois sur le Web sous forme de page HTML mais également sous un format plus imprimable, comme PDF.

Publication de graphismes

La publication de graphismes est un autre aspect de la publication. Il n'est pas rare que des données soient liées à une dimension temporelle et nécessitent alors un affichage sous forme d'histogramme, de camembert… On peut, par exemple, imaginer des cotations boursières stockées en XML, ou bien des automates délivrant des informations dont il faut faire la synthèse. L'affichage peut résulter de différentes méthodes. Une première méthode consiste à recréer une image côté serveur (aux formats PNG, JPEG…), cette solution présentant l'avantage d'être facilement intégrable à l'existant, mais ayant l'inconvénient de limiter les interactions de l'utilisateur sur la courbe (info-bulle sur les points principaux…). Une autre méthode consiste à utiliser un format vectoriel côté client, l'application cliente calculant alors le graphe et proposant des services supplémentaires (redimensionnement, zoom, info-bulle…). Parmi les formats de ce type, nous pouvons retenir SVG (*Scalable Vector Graphics*).

Le format pour le Web : XHTML

XHTML (*Extensible HyperText Markup Language)* reprend le formalisme du langage HTML (*Hypertext Markup Language*) en transposant l'origine SGML (*Standard Generalized Markup Language*) par une parenté XML. XHTML est donc un langage XML qui suit la syntaxe des documents XML (plus contraignante que SGML). La dernière version officielle est la version 1.1 mais la 2.0 est en cours de préparation au moment de la rédaction de cet ouvrage. Il existe deux types de DTD pour XHTML :

- Les DTD de transition, qui gardent une certaine compatibilité avec les versions antérieures.
- Les DTD strictes, qui ne contiennent que ce que prévoit le standard et ne facilitent donc pas les transitions avec un existant.

Dans l'idéal, XHTML est orienté structure, c'est-à-dire qu'il décrit des relations entre différents blocs de texte (titre, paragraphe…), la présentation étant théoriquement dévolue aux feuilles de styles (CSS).

La connaissance des balises et des attributs principaux de ce format est un minumum indispensable pour réaliser de la publication dans un univers intranet/Internet. Les balises XHTML appartiennent à l'espace de noms *http://www.w3.org/1999/xhtml*.

L'objectif de cette partie n'est pas de décrire toutes les possibilités de ce langage, mais au moins de vous donner quelques pistes pour commencer à travailler avec.

Les principales balises de XHTML

Un document XHTML suit ce type de structure :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body>...
</body>
</html>
```

La balise `html` est donc englobante ; la balise `head` est optionnelle et comportera des informations plus génériques, comme le titre de la page ou des champs `meta`, principalement pour les moteurs de recherche… La balise `body` comportera toute la partie visible de la page.

Nous pouvons désigner la langue employée par l'attribut `xml:lang` sur la balise `html` (`fr` pour français).

Les balises d'en-tête

L'en-tête est rarement absent d'une page, ne serait-ce que pour le titre du document. Il contient souvent une référence à un ou plusieurs fichiers CSS voire des scripts JavaScript. On peut également lui assigner des champs `meta`, suivant deux structures possibles : la première sous la forme `<meta name="Nature" content="Attribut"/>` et la deuxième sous la forme `<meta http-equiv="…" content="…"/>`. Cette dernière forme est utilisée pour les redirections. Concernant la première forme, nous trouvons les natures suivantes :

- `author` : informations sur l'auteur de la page ;
- `copyright` : référence des informations de droits d'auteur ;
- `description` : information à afficher lors du résultat d'une recherche ;
- `expires` : indique au robot la date d'expiration de la page ;
- `generator` : nom de l'éditeur HTML ayant généré la page web ;
- `keywords` : mots-clés décrivant la page web ;
- `robots` : principalement pour le moteur Google ;
  - `all` : par défaut (tout indexer) ;
  - `follow` : suivre les liens relatifs ;
  - `index` : indexer la page ;

  - `nofollow` : ne pas analyser les pages liées ;
  - `noindex` : ne pas indexer la page ;
  - `none` : ne rien faire.
- `revisit-after` : délais de visite par le robot ;
- `subject` : sujet de la page.

Les balises de structure

Ces balises de structure n'ont de sens que dans la balise `body`.

Nous disposons des balises d'en-tête `h1` à `h6`, `h1` étant le niveau le plus important. La structuration en paragraphes se fait par la balise `p`.

> **Retour à la ligne**
>
> Les retours à la ligne, dans le texte, sont supprimés par le navigateur : il décide, selon l'espace disponible, d'effectuer d'autres retours à la ligne. Cependant, il est parfois incontournable de placer un retour à ligne à un endroit précis et nous utiliserons alors la balise auto-fermée `br`. À noter qu'un certain nombre de balises (en-tête, paragraphe…) occasionnent au moins un retour à la ligne.

Les puces sont réalisées par les balises `OL` (*Ordered List*) ou `UL` (*Unordered List*). Chaque puce est représentée par la balise `LI` (*List Item*).

Exemple dans la balise `body` :

```
<h1>1. Titre</h1>
<h2>a. Sous titre</h2>
<p>Un paragraphe .... </p>
<p>Choix :
<ul>
 <li>Choix 1</li>
 <li>Choix 2</li>
</ul>
</p>
```

La balise auto-fermée `hr` est une autre balise de structure qui permet de réaliser une ligne horizontale.

La représentation des liens

Les liens vers une ressource (nouvelle page, image…) peuvent être internes ou externes. Un lien est défini par la balise `a` avec pour attribut `href`.

Exemple d'un lien vers une ressource externe :

```
<a href="autrepage.html">Aller vers une autre page</a>
```

Lorsque un lien fait référence à une partie de la page en cours, on positionne une ancre, via le même élément `a`, avec un attribut `name`. Le lien s'effectue alors en préfixant le nom de l'ancre par un caractère #.

Voici un exemple pour se positionner à la fin de la page :

```
<a  href="#bas">Aller à la fin de la page</a>
…
<a name="bas"/>Fin de page…
```

Exercice 1

Gestion des liens

Créez une page `tp1.html` contenant une table des matières avec des liens internes vers différentes parties de la page (`Titre1`, `Titre2`…). Chaque partie contiendra également un texte et un lien vers une page externe. Vous utiliserez des puces pour afficher la table des matières.

L'intégration d'images

Les images sont insérées par la balise auto-fermée `img`. L'attribut `src` contient une URL relative ou absolue vers la ressource image, celle-ci étant principalement aux formats PNG (*Portable Network Graphics*), JPEG ou GIF.

Quelques attributs de la balise `img` :

- `alt` : texte affiché si l'image ne peut être rendue visible ; utile également pour les personnes ayant un handicap visuel (synthèse vocale du contenu).
- `width`, `height` : taille de l'image ; en règle générale, il vaut mieux redimensionner l'image dans un logiciel de dessin pour éviter le passage sur le réseau de données inutiles.
- `border` : affichage d'une bordure ; on donne une taille en pixels.
- `hspace` / `vspace` : écart horizontal et vertical avec le texte adjacent.

À une image il est possible d'associer une `MAP`. Cette dernière est composée d'un ensemble de zones de nature circulaire, rectangulaire ou polygonale, chaque zone se comportant comme un lien.

Voici un exemple d'image (carte de France) avec deux zones réactives (2 régions) :

```
<map name="MAP1">
 <area href="IDF.html" alt="Ile de france" coords="150,150,95,195"/>
 <area href="bretagne.html" alt="Bretagne" coords="15,5,135,50">
</map><img src="france.png" alt="Site map" usemap="#MAP1"/>
```

On comprendra donc que chaque zone est associée à la balise `area`. Le lien entre l'image et l'ensemble des zones s'effectue par l'attribut `usemap` (ne pas oublier d'ajouter un caractère #).

L'insertion des tableaux

Les tableaux sont d'un usage courant. Ils sont d'autant plus importants qu'un document XML comportant des structures répétitives (liste de personnes dans un annuaire, par exemple) se représente habituellement sous cette forme.

Les tableaux peuvent également servir à découper une page en zones (notion de *templates* ou modèles de page) en assurant un alignement du texte. C'est aussi le cas pour les formulaires qui sont plutôt présentés de manière tabulaire.

Un tableau est décomposé en lignes (balise `tr` pour `table row`). Chaque ligne est divisée en colonnes (balise `td` pour `table data`). Il existe également une balise pour le titre (titre de ligne ou de colonne) appelé `th` (`table header`).

Pour représenter une cellule vide, on insère un blanc spécial via l'entité `nbsp` (`non-breaking space`).

Voici un exemple de table avec trois lignes et deux colonnes, la première ligne contenant le titre des colonnes :

```
<table>
<tr>
 <th>Colonne 1</th>
 <th>Colonne 2</th>
</tr>
<tr>
 <td>10</td>
 <td>20</td>
</tr>
<tr>
 <td>30</td>
 <td>40</td>
</tr>
</table>
```

Il est également possible de délimiter certaines portions du tableau comme l'en-tête, les parties et la fin, par les balises `thead`, `tbody` et `tfoot`.

Voici un autre exemple :

```
<table>
<thead>
 <tr>
  <th>Titre 1</th>
  <th>Titre 2</th>
 </tr>
</thead>
<tfoot>
 <tr>
  <td>Fin 1</td>
  <td>Fin 2</td>
 </tr>
</tfoot>
```

```
<tbody>
 <tr>
  <td>1</td>
  <td>2</td>
 </tr>
</tbody>
</table>
```

Voici maintenant quelques attributs du tableau :

- `background` : image d'arrière-plan ;
- `bgcolor` : couleur de fond ;
- `border` : épaisseur de la bordure ;
- `cellpadding` : marge dans les cellules ;
- `cellspacing` : espace entre les cellules ;
- `summary` : résumé du contenu (accessibilité) ;
- `title` : titre du tableau (accessibilité) ;
- `width` : largeur du tableau en valeur absolue ou pourcentage (la valeur est alors suivie du caractère `%`).

Quelques attributs de la ligne :

- `align` : alignement horizontal (`left`, `center`, `right`, `justify`) ;
- `bgcolor` : couleur de fond ;
- `height` : hauteur de la ligne (en pixels) ;
- `valign` : alignement vertical (`bottom`, `middle`, `top`).

Quelques attributs de la cellule (`th` ou `td`) :

- `align` : alignement horizontal (`left`, `center`, `right`, `justify`) ;
- `background` : image d'arrière-plan ;
- `bgcolor` : couleur de fond ;
- `colspan` / `rowspan` : fusion de colonnes ou de lignes ; la valeur est un entier ;
- `valign` : alignement vertical (`bottom`, `middle`, `top`) ;
- `width` : largeur de la cellule (elle devrait être la même sur toutes les lignes) en valeur absolue ou en pourcentage (la valeur est alors suivie du caractère `%`).

Voici un autre exemple de tableau avec fusion des deux colonnes pour la dernière ligne :

```
<table width="90%" border="1" cellspacing="3" cellpadding="2">
 <tr>
  <td width="63%">Bonjour</td>
  <td width="37%" bgcolor="#006699">Au revoir</td>
 </tr>
```

```
 <tr>
  <td colspan="2">Hello World </td>
 </tr>
</table>
```

Exercice 2

Utilisation de tableaux

Réalisez un tableau de trois colonnes et quatre lignes. Chaque cellule contiendra un numéro et la diagonale du tableau sera de couleur rouge. La dernière ligne sera composée d'une cellule occupant les trois colonnes.

Les balises DIV et SPAN : conteneurs génériques

Certaines balises jouent un peu le rôle de boîtes à tout faire. Les balises `div` et `span` font partie de cette catégorie, la première agissant sur un contenu multiligne, alors que la seconde est utilisée pour des portions d'une ligne.

À ces conteneurs génériques, on associe souvent un attribut `id` qui a un type `ID` dans la DTD et n'autorise donc qu'une valeur unique. Cela a l'avantage d'être facilement exploitable dans un langage de script (comme JavaScript) ou pour appliquer des propriétés graphiques par CSS (délimitation de l'application des propriétés par l'`id`).

Voici un exemple de conteneur `div` que l'on positionne en absolu :

```
<div style="position:absolute;left: 50px;top:50px;width:700px;height:
➥400px;">Hello World</div>
```

Les feuilles de styles : le langage CSS

CSS (`Cascading Style Sheets`) est un langage indépendant de XHTML/HTML. Il n'est pas construit à partir de balises mais s'appuie sur un mécanisme de filtrage de balises et d'application de propriétés. Le langage CSS peut également s'appliquer à tous les documents XML. Cependant, sa capacité à réaliser des transformations de structure étant très limitée, on lui préfère les transformations XSLT (*XSL Transformations*). Ces transformations aboutissent souvent à la création d'un document XHTML auquel on associe une ou plusieurs feuilles de styles CSS.

On le retrouve également auprès de certains éditeurs XML qui tentent de gommer la syntaxe XML au profit d'une édition plus visuelle (de façon analogue à un éditeur comme Word).

Certains langages de présentation, comme SVG (*Scalable Vector Graphics*) ou XSL-FO (*Extensible Stylesheet Language-Formatting Objects*), s'en inspirent également, notamment en termes de propriétés graphiques.

Le langage CSS possède trois versions : *level* 1 à 3. La version 2 étant la plus courante et la version 3 étant encore peu supportée par les navigateurs (*http://www.w3.org/Style/CSS/current-work*).

Nous n'allons pas passer en revue la totalité des propriétés CSS (il y en a une bonne centaine) mais au moins en cerner le fonctionnement pour vous rendre opérationnel.

Origine du mot cascade

On parle de feuilles de styles en cascade car il existe plusieurs niveaux d'application et certaines règles peuvent être partiellement en conflit. Le navigateur applique alors un algorithme pour recalculer la somme des propriétés CSS applicables. En règle générale, on peut retenir que plus on est précis et plus la priorité augmente.

Voici les différents niveaux d'application des styles CSS, de la priorité la plus faible à la priorité la plus forte :

- le style par défaut (choix du navigateur) ;
- un fichier CSS ;
- un contenu CSS interne à la page HTML (*header* de la page) ;
- un contenu CSS interne à une balise HTML.

Différentes déclarations des styles CSS dans une page XHTML ou HTML

Les propriétés graphiques décrites en CSS sont applicables à différents niveaux. Le cas le plus simple et ayant toujours priorité est la déclaration des propriétés dans une balise XHTML via l'attribut `style`. Les propriétés se présentent sous la forme `clé:valeur` et sont séparés par un `;`.

Voici un exemple sur un paragraphe :

```
<p style="color:red;font-style:italic">...</p>
```

Dans ce paragraphe, nous avons spécifié que le texte devait être en rouge et en italique.

Une autre possibilité d'usage est l'insertion du langage CSS dans l'en-tête de la page HTML à l'aide de la balise `style`. Bien sûr, on ne peut pas se contenter de définir des propriétés graphiques dans ce contexte : il faut également préciser sur quelles portions de la page nous allons les appliquer.

Voici un exemple :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <style type="text/css">
   p { color :red; font-style:italic; }
  </style>
 </head>
</html>
```

Ici, nous avons repris l'exemple ci-avant sur la balise paragraphe en le généralisant à toutes les balises paragraphe.

Enfin, il est possible de décrire les styles CSS dans un fichier indépendant et d'insérer, dans la page XHTML, un lien vers ce fichier. On l'aura compris, ce système présente un avantage certain pour la maintenance d'un site, car il n'y a plus besoin de passer dans chaque page pour changer des caractéristiques graphiques (couleurs…).

Soit le fichier `monFichier.css` :

```
p {
 color : red;
 font-style : italic;
}
```

Le lien vers ce fichier CSS se fait avec la balise `link` de cette manière :

```
<html>
<head>
 <link rel="stylesheet" href="monFichier.css"/>
</head>
...
</html>
```

Pour un document XML, on indiquera le lien vers la feuille de styles avec l'instruction de traitement `xml-stylesheet` :

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="affichage.css"?>
<annuaire>
…
</annuaire>
```

Le rôle de l'attribut média

La feuille de styles peut être chargée uniquement dans certains contextes d'usage lorsqu'on précise l'attribut `media` dans les balises `link` ou `style`.

Voici les valeurs possibles :

- `screen` : écran graphique ;
- `tty` : terminal (uniquement des caractères) ;
- `tv` : télévision, faible résolution ;
- `projection` : projecteur ;
- `handheld` : PDA ;
- `print` : impression ;
- `braille` : appareil tactile ;
- `aural` : synthétiseur vocal ;
- `all` : tous les appareils.

Exemple :

```
<link media="print,braille" rel="stylesheet" href="fichier-
➥Impression.css">
```

Dans cet exemple, le fichier `fichierImpression.css` n'est chargé que lors d'une impression ou lors de l'usage d'un appareil pour non-voyant.

La syntaxe d'une feuille de styles

Une règle CSS suit cette syntaxe :

```
selecteur { ensemble de propriétés : valeur }
```

Le sélecteur sert à délimiter le champ d'application des propriétés, le cas le plus simple étant le nom d'une balise, ce qui signifie que les propriétés s'appliqueront sur toutes les occurrences de la balise.

Voici un exemple :

```
body {
background-color:black;
color:white;
}
p {
border-style:double;
}
```

Pour la balise page (`body`), un fond noir et une encre blanche seront utilisés. La bordure de la balise paragraphe (`p`) sera composée d'une double ligne.

On peut également grouper les balises par une virgule.

Par exemple :

```
p,div,h1 { color:red; }
```

Ici les balises `p`, `div` et `h1` auront une encre rouge.

Pour appliquer des propriétés graphiques à un sous-ensemble d'occurrences, on positionne un attribut `class`. La valeur de cet attribut peut ensuite être reprise dans le sélecteur CSS.

Par exemple dans cette page XHTML, nous pouvons écrire :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
 <p class="introduction">
 Bonjour
 </p>
 <p class="conclusion">
 Au revoir
 </p>
 <p class="introduction conclusion">
 ...
 </p>
</body>
</html>
```

Les premier et dernier paragraphes appartiennent à la classe `introduction`, alors que celui du milieu et le dernier appartiennent à la classe `conclusion`.

Pour appliquer des propriétés CSS dans ces différentes zones, on sépare la balise de la classe par un point.

Exemple :

```
p.introduction { color:green; }
p.conclusion { background-color:blue; }
```

Une encre verte sera utilisée pour les premier et dernier paragraphes, alors que le deuxième et le dernier auront une couleur de fond bleue.

On peut également se focaliser sur la classe plutôt que sur la balise de cette manière :

```
.stylecommun {
 color:red;
 background-color:gray;
}
```

N'importe quelle balise peut appartenir à la classe `stylecommun`, ce qui produira un affichage avec une encre rouge et une couleur de fond grise.

Voici un exemple de page XHTML correspondant :

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<p id="monId" class="stylecommun">
Bonjour
</p>
<div class="stylecommun">
Au revoir
</div>
</body>
</html>
```

Une autre possibilité de sélecteur consiste à spécifier un identifiant préfixé par le caractère # :

```
p#monId {
color:white;
}
```

Cet exemple permet d'appliquer une encre blanche à un paragraphe dont l'attribut `id` a pour valeur `monId`. Il est également possible d'ignorer la balise en employant un identifiant de cette manière :

```
#monId {
 color:white;
}
```

Une contrainte de descendance peut être précisée par un blanc séparant plusieurs sélecteurs :

```
div p {
color : red;
}
```

Dans cet exemple, tous les paragraphes contenus dans une balise `div` seront affichés avec une encre rouge.

Nous présentons ci-après d'autres formes de sélecteur, mais qui sont encore mal supporté par certains navigateurs (notamment Internet Explorer) :

- `A > B` si `A` est parent de `B`.
- `A + B` si `A` et `B` possèdent un parent commun et se suivent.
- `A[P]` si l'élément `A` possède l'attribut `P`.
- `A[P=V]` si l'élément `A` possède l'attribut `P` avec la valeur `V`.

Il existe une dernière forme de sélecteur plus contextuel fonctionnant, par exemple, sur les liens selon l'état du lien (actif, activé, visité, survolé)…

Voici quelques états possibles :

- `link` : lien par défaut ;
- `visited` : lien visité ;
- `hover` : survol d'un lien ;
- `active` : lien activé ;
- `first-letter` : première lettre d'un texte ;
- `first-line` : première ligne d'un texte ;
- `first-child` : premier fils ;
- `last-child` : dernier fils.

Exemple d'utilisation :

```
a:link {
color:yellow;
}
a:hover {
color:red;
}
```

Dans ce cas, le lien est coloré en jaune par défaut et devient rouge lors du passage de la souris.

Quelques propriétés CSS

Nous présentons ici les principales propriétés CSS et leur signification. Elles ont été triées par catégorie.

Le cas des polices de caractères

- `font` : regroupe directement les différentes propriétés de la police ;
- `font-family` : définit la famille de la police ;
- `font-size` : définit la taille de la police ;
- `font-style` : définit l'orientation de la police ;
- `font-variant` : affiche la police en petites majuscules ;
- `font-weight` : définit la graisse de police.

Type de police

Les polices sont regroupées en deux catégories : avec ou sans empattement (serif). L'empattement est la présence de segments supplémentaires aux extrémités de chaque lettre. On choisit plutôt des polices sans empattement à l'écran et pour l'impression. Exemple type de police à l'écran : `Arial` ; en impression : `Times New Roman`. Les polices sont généralement proportionnelles, c'est-à-dire que les lettres ont des largeurs différentes. Il est cependant possible d'utiliser la police `Courier` pour obtenir des espaces homogènes.

Exemple :

```
Exemple :
.titre {
  font: bold 10px Times ;
}
```

Le cas des couleurs et du fond

- `background` : regroupe les différentes propriétés CSS permettant de définir un arrière-plan ;
- `background-attachment` : fige une image d'arrière-plan ;
- `background-color` : définit la couleur de fond ;
- `background-image` : affiche une image en arrière-plan ;
- `background-position` : positionne une image d'arrière-plan ;
- `background-repeat` : limite et contrôle la répétition d'une image d'arrière-plan ;
- `color` : couleur du texte.

Les couleurs

Les couleurs s'expriment par la combinaison des intensités (de 0 à 255) sur les composantes rouge, verte et bleue, le noir étant l'intensité minimale pour chaque composante et le blanc l'intensité maximale. Une couleur s'exprime généralement en hexadécimal selon la syntaxe `#RRVVBB` (`R` pour rouge, `V` pour vert et `B` pour bleu). Chaque composante peut donc s'exprimer par une valeur comprise entre `00` et `FF`. Par exemple `#00FF00` signifie un vert intense (seule la deuxième composante est active). Certaines couleurs sont aussi prédéfinies.

Quelques couleurs prédéfinies selon la DTD *http://www.w3.org/TR/xhtml1/dtds.html*.

```
Black  = #000000    Green  = #008000
Silver = #C0C0C0    Lime   = #00FF00
Gray   = #808080    Olive  = #808000
White  = #FFFFFF    Yellow = #FFFF00
Maroon = #800000    Navy   = #000080
Red    = #FF0000    Blue   = #0000FF
Purple = #800080    Teal   = #008080
Fuchsia= #FF00FF    Aqua   = #00FFFF
```

Exemple :

```
.maPage {
background : url(fond.gif) no-repeat 50% 50% ;
}
```

Lien vers une ressource

En CSS, les valeurs sont en quelque sorte typées par des fonctions. L'URL nécessite une fonction `url` comme dans l'exemple ci-dessus.

Le cas du texte

- `letter-spacing` : définit l'espace entre les caractères ;
- `line-height` : définit l'interligne dans un bloc de texte ;
- `text-decoration` : personnalise l'apparence d'un texte ;
- `text-align` : aligne horizontalement le contenu d'un élément ;
- `text-indent` : crée un alinéa pour la première ligne d'un texte ;
- `text-transform` : définit les effets de capitalisation d'un texte ;
- `text-shadow` : effet 3D ;
- `vertical-align` : définit l'alignement vertical ;
- `white-space` : gère l'affichage des blancs et la césure ;
- `word-spacing` : définit l'espace entre les mots d'un texte.

Exemple :

```
.note {
text-transform : uppercase ;
text-align : center ;
}
```

- `border` : regroupe les différentes propriétés CSS pour définir les bordures ;
- `border-color` : couleur de la bordure ;
- `border-style` : permet de définir le type des bordures ;

- `border-width` : épaisseur de la bordure ;
- `border-top` : différentes propriétés CSS pour la bordure en haut ;
- `border-right` : différentes propriétés CSS pour la bordure à droite ;
- `border-bottom` : différentes propriétés CSS pour la bordure en bas ;
- `border-left` : différentes propriétés CSS pour la bordure à gauche ;
- `border-top-color` : couleur de la bordure en haut ;
- `border-right-color` : couleur de la bordure à droite ;
- `border-bottom-color` : couleur de la bordure en bas ;
- `border-left-color` : couleur de la bordure à gauche ;
- `border-top-style` : type de la bordure en haut ;
- `border-right-style` : type de bordure à droite ;
- `border-bottom-style` : type de bordure en bas ;
- `border-left-style` : type de bordure à gauche ;
- `border-top-width` : épaisseur de bordure en haut ;
- `border-right-width` : épaisseur de bordure à droite ;
- `border-bottom-width` : épaisseur de bordure en bas ;
- `border-left-width` : épaisseur de bordure à gauche ;
- `height` : hauteur d'un élément ;
- `margin` : différentes propriétés permettant de définir les marges externes ;
- `margin-top` : marge haute ;
- `margin-right` : marge à droite ;
- `margin-bottom` : marge en bas ;
- `margin-left` : marge à gauche ;
- `max-height` : hauteur maximale ;
- `max-width` : largeur maximale ;
- `min-height` : hauteur minimale ;
- `min-width` : largeur minimale ;
- `padding` : différentes propriétés permettant de définir les marges internes ;
- `padding-top` : espace intérieur en haut ;
- `padding-right` : espace intérieur à droite ;
- `padding-bottom` : espace intérieur en bas ;

- `padding-left` : espace intérieur à gauche ;
- `vertical-align` : alignement vertical ;
- `width` : largeur.

Exemple :

```
.padding1 {
  padding : 30px 50px 5px 20px;
}
```

Le cas du mode de positionnement

- `clear` : détermine si un élément peut se trouver sur la même bande horizontale qu'un élément flottant (image...). Cela évite qu'une partie de l'élément affichée soit recouverte.
- `display` : contrôle l'affichage des éléments (en ligne, bloc...).
- `float` : spécifie de quel côté du conteneur (gauche, droite...), l'élément à afficher doit être aligné.
- `position` : déterminer l'emplacement de tel ou tel élément.
- `top` : position haute.
- `right` : position droite.
- `bottom` : position basse.
- `left` : position gauche.
- `visibility` : cache ou non certains éléments.
- `z-index` : superposition des éléments en définissant l'ordre d'empilement.

Exemple :

```
.texte{
  display : block;
}
```

Les cas restants

- `list-style-type` : définit le style (ou l'apparence) de la puce ;
- `list-style-image` : définit l'image qui sera utilisée comme puce ;
- `list-style-position` : détermine le retrait de la puce ;
- `list-style` : regroupe les différentes propriétés CSS pour définir une liste ;
- `cursor` : change le curseur lors d'un survol ;
- `size` : sert à définir la taille de la page imprimée ;
- `page-break-before` : spécifie qu'un élément est précédé d'un saut de page ;
- `page-break-after` : spécifie qu'un élément est suivi d'un saut de page.

Exemple :

```
ul {
  list-style-image: url('puce.gif');
}
```

Exercice 3

Utilisation du CSS

Reprenez le document de l'exercice 1 et appliquez-lui une feuille de styles externe. Les textes utiliseront une police Arial. Les titres principaux seront en rouge et italique et les autres textes en vert. La couleur du lien changera lors du survol de la souris. Les puces seront remplacées par des images.

L'application des CSS à un document XML

L'application directe des styles CSS à un document XML est possible mais limitée. Cela n'est pas le cas dans la page XHTML car des balises, même celles de structure, sont déjà liées à un contexte de publication de texte (en-tête, paragraphe…).

Soit l'extrait suivant d'un document XML :

```
<carnet>
<personne>
 <nom>Dupont</nom>
 <age>44</age>
</personne>
<personne>
 <nom>Dupond</nom>
 <age>43</age>
</personne>
</carnet>
```

Si nous souhaitons définir un fichier CSS pour présenter ce document de manière rudimentaire, nous pouvons écrire le fichier `carnet.css` suivant :

```
nom {
display :block ;
color :blue ;
}
age {
color :red ;
}
```

La propriété `display` va servir à insérer des retours à la ligne. Chaque `personne` sera ainsi associée à un paragraphe. Le lien vers le fichier CSS sera présent via l'instruction de traitement suivante, placée avant la racine du document XML :

```
<?xml-stylesheet type="text/css" href="carnet.css"?>
```

Le langage de requête XPath

XPath est un langage dont la syntaxe n'a rien à voir avec celle du langage XML. Son but principal est d'exprimer des requêtes pour localiser des parties d'un document XML. Les réponses à ces requêtes seront ensuite introduites dans un *template* résultat (souvent en XHTML/HTML) dans les documents XSLT. Son champ d'application est cependant plus vaste : XPath peut, par exemple, servir avec les bases de données qui supportent XML en natif, ou bien dans un cadre plus courant d'applicatif.

XPath a cependant une limitation : il est difficilement employable sans une représentation globale du document. D'ailleurs les API connues passent par une représentation DOM ou un équivalent (modèle objet d'une arborescence XML) pour effectuer l'exécution.

La version 1.0 de XPath

La version 1.0 de XPath est presque incontournable ; on la retrouve également dans la version 2.0. Elle est présente comme sous-ensemble dans l'expression des clés et des relations de clés dans les schémas W3C et s'utilise pleinement dans certaines instructions XSLT.

Quelques points de vocabulaire

Un document XML est une arborescence. Cette arborescence est composée de nœuds de différentes catégories, les catégories principales étant l'élément (la balise) et le texte.

Pour effectuer une recherche, nous pouvons partir de la racine (chemin absolu) ou bien préciser un point de départ, que l'on nomme parfois nœud de référence.

Lorsqu'on parcourt un arbre afin d'obtenir une ou plusieurs valeurs, on peut passer par des nœuds intermédiaires, appelés nœuds de contexte.

Un résultat composé de plusieurs nœuds élément est un *NodeSet*. Il est par définition non ordonné.

Les caractéristiques de XPath 1.0

Voici quelques caractéristiques de la version 1.0 :

- XPath est principalement un langage de requêtes dans un arbre XML (on peut réaliser une opération telle que 1+1 mais ce n'est pas le but principal).
- Il est fondé sur des chemins d'accès, ou chemins de localisation. On peut faire l'analogie avec un chamin (*path*) vers un fichier ou bien avec une suite de déplacements pour arriver à une destination (2e à gauche…).
- Il fonctionne en chemin relatif ou absolu, l'absolu étant toujours le passage par la racine. Une erreur classique consiste à effectuer une requête absolue qui fait perdre le contexte courant.
- Il s'applique à des nœuds, des booléens, des nombres et chaînes de caractères.

- Une bibliothèque de fonctions est disponible.
- Le résultat produit est un *NodeSet*, un nombre décimal, une chaîne ou un booléen
- Il existe sous une forme abrégée et non abrégée, les deux formes n'étant pas équivalentes. La forme non abrégée couvre plus de possibilités.
- L'opérateur | représente l'union de plusieurs expressions XPath.

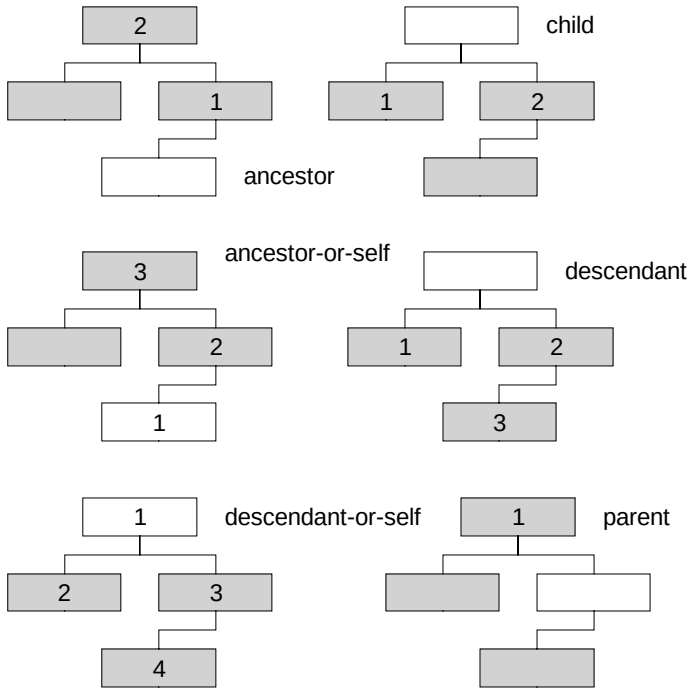
La notion de chemin de localisation

Un chemin de localisation sert à désigner une partie de l'arborescence XML. Il est composé de :

- un Axe : direction dans l'arbre (les fils, les frères, les ancêtres...) ;
- un type de nœud à localiser : s'applique aux nœuds selon l'axe (les éléments, les textes...) ;
- 0 à n prédicats : des conditions à respecter (expression booléenne à vérifier sur chaque nœud trouvé).

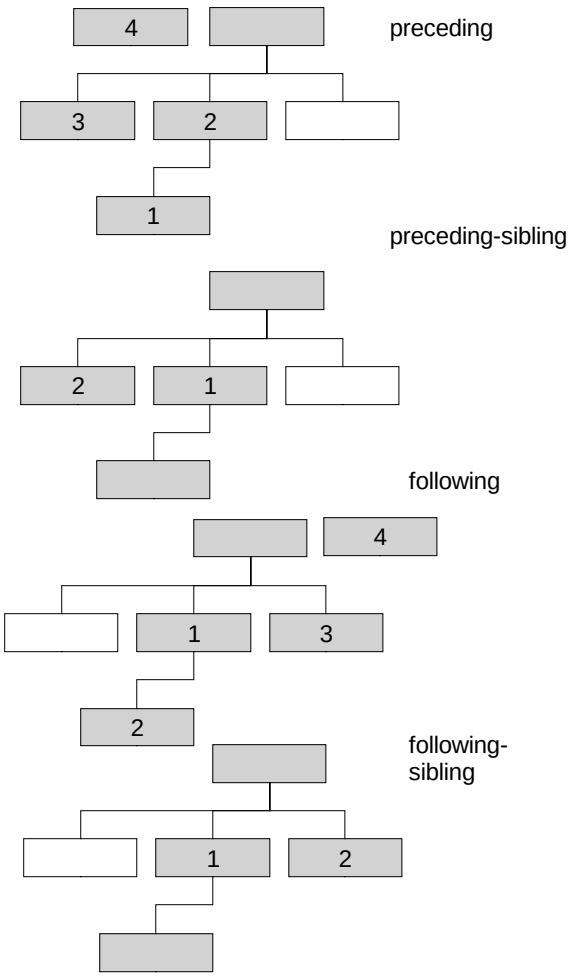
Les deux premiers composants sont obligatoires alors que le dernier est optionnel. On peut considérer que ces trois composants agissent sur trois niveaux différents pour réaliser la localisation de nœuds, l'axe étant le moins précis, puisqu'il n'indique qu'une direction, et le prédicat étant le plus précis, pour ne retenir que certains nœuds.

Figure 5-1
Principaux axes



Dans la figure 5-1, l'axe est désigné en anglais. Le rectangle blanc correspond au point de départ. Les numéros correspondent à l'ordre de la réponse (le premier nœud…). Si nous prenons l'axe courant `child`, on constate que le parcours correspond à la recherche des descendants d'un nœud de premier niveau (les fils).

Figure 5-2
Autres axes



La figure 5-2 contient des axes probablement moins employés. Pour comprendre les rôles de `preceding` (précédent) et `following` (suivant), il faut imaginer une découpe en deux parties de l'arborescence, suivant une ligne de séparation composée du nœud de référence et de ces ancêtres (parent, parent de parent…).

Il existe d'autres types d'axes un peu plus spécifiques qui jouent davantage le rôle de filtrage que de localisation, comme `attribute` et `namespace`. Ils agissent pour extraire des nœuds liés à l'élément de contexte.

L'axe `self` représente le nœud courant. Bien qu'à première vue peu utile, il peut servir à vérifier la nature du nœud de contexte ou bien à réaliser des prédicats plus sophistiqués (plusieurs contraintes sur le nœud courant que l'on désigne par `self`).

Les types de nœuds

Pour désigner un nœud, la forme la plus simple est le nom de l'élément (la balise). `*` est un raccourci pour désigner tous les éléments ou bien tous les attributs selon l'axe employé.

Il existe ensuite un ensemble de fonctions qui servent à reconnaître chaque type de nœud, comme :

- `text()` : uniquement les nœuds textes ;
- `node()` : tous les nœuds ;
- `comment()` : les commentaires ;
- `processing-instruction()` : les instructions de traitement.

Voici maintenant quelques exemples XPath au format non abrégé combinant axes et types de nœud (noter bien la séquence `::` pour séparer) :

```
child::para
```

Tous les éléments fils de nom `para`.

```
child::*
```

Tous les éléments fils.

```
descendant::*
```

Tous les éléments descendants.

```
attribute::titre
```

L'attribut `titre` du nœud de contexte (ou référence).

```
self::racine
```

Le nœud `racine` courant ou un ensemble vide si cela ne correspond pas.

Un cheminement (comme pour un *path* vers un fichier) est découpé en plusieurs parties. Cela permet d'effectuer un parcours par niveau, le caractère `/` séparant chaque niveau.

L'exemple suivant :

```
child::personne/child::email
```

permet de rechercher l'ensemble des éléments `email` dans les nœuds `personne` du nœud de référence.

```
/child::carnet/descendant::email
```

L'exemple ci-avant permet de rechercher l'ensemble des éléments `email` en partant de la racine `carnet`.

Ces deux exemples sont un peu différents. Le caractère *slash* (`/`), situé au début du deuxième exemple, signifie que l'on commencera toujours par la racine du document (accès absolu) ; le nœud de référence n'aura alors pas d'utilité.

L'augmentation de la puissance des requêtes par les prédicats

Les prédicats n'ont pas pour objectif de retourner un résultat sous la forme d'un ensemble de nœuds, mais d'exprimer une condition avec un état vrai ou faux. Si l'état est vrai, le nœud courant fera partie du résultat. Le prédicat suit le type de nœud dans un chemin de localisation et est décrit entre crochets. Le nœud courant sera alors retenu comme faisant partie de la réponse si cette expression, sous forme de prédicat, a une réponse. Une réponse qui ne sera pas booléenne sera convertie (conversion implicite). Par exemple, un ensemble de nœuds non vides sera associé à la valeur vraie ; inversement un ensemble de nœuds vides sera associé à la valeur fausse.

Le prédicat peut représenter un chiffre (commençant par 1) et désigne le numéro du nœud selon l'ordre de parcours. Des comparaisons (égalité, ordre de grandeur…) entre expressions sont possibles par les opérateurs `<`,`<=`,`>`,`>=`,`=`. À noter que la négation s'exprime par la fonction `not`. Les opérateurs `or` et `and` servent également aux expressions booléennes. Enfin, l'opérateur `mod` retourne le reste de la division entière (modulo).

Un prédicat peut contenir une autre expression XPath, sachant que le nœud de référence de cette expression sera le nœud courant (ou de contexte) trouvé par la partie gauche de l'expression. Si cette expression retourne un ensemble de nœuds non vides, le prédicat rendra valide le nœud en cours pour la réponse finale.

Lorsque des opérateurs (comme l'égalité) s'appliquent sur des valeurs de type différent, une conversion implicite est réalisée. Par exemple, si nous comparons un nœud avec un texte, la seule conversion possible sera de trouver la forme textuelle du nœud (son contenu texte), afin de bien comparer deux textes au final. Ces conversions implicites peuvent parfois être trompeuses. La conversion d'un nœud en texte, par exemple, va consister à concaténer tous les textes présents dans ce nœud (même dans les éléments descendants).

Voici quelques exemples d'expressions XPath avec prédicat :

```
child::personne[child::nom='dupont']
```

L'expression ci-avant permet de rechercher tous les éléments fils `personne` dont un fils `nom` contient le texte `dupont`.

```
child::nom[self::nom='dupond']
```

Cette expression permet de rechercher tous les éléments fils `nom` dont le texte contient `dupond`.

```
child::personne[attribute::numero='10']
```

Recherche de tous les éléments fils `personne` dont l'attribut `numero` a pour valeur 10. Si nous n'avions pas mis des apostrophes (*quotes*) simples, une conversion en nombre entier de la valeur de l'attribut aurait été réalisée (`010` aurait donc aussi fonctionné, par exemple).

```
child::personne[attribute::numero]
```

Recherche de l'ensemble des éléments fils `personne` ayant un attribut `numero`.

```
child::personne[child::nom or child::prenom]
```

Recherche de l'ensemble des éléments fils `personne` ayant au moins un élément `nom` ou un élément `prenom`.

```
child::personne[not(attribute::*)]
```

Recherche de l'ensemble des éléments fils `personne` sans attributs.

Utilisation des fonctions avec XPath

Les fonctions XPath sont nombreuses et sont exploitables soit dans un prédicat, soit directement comme expression XPath.

Voici quelques fonctions :

- `string-length( ... )` : longueur d'une chaîne ;
- `starts-with( chaîne1, chaîne2 )` : tester si `chaîne1` commence par `chaîne2` ;
- `substring( chaîne1, position1, longueur)` : extraction d'une sous-chaîne ;
- `normalize-space( chaîne )` : normalisation des occurrences de blancs à 1 blanc ; suppression des blancs d'en-tête et de fin ;
- `translate( chaîne, caractères source, caractères destination )` : convertit dans la `chaîne` tous les `caractères source` par leur correspondance (en fonction de la position) dans le dernier argument ;
- `number( chaîne )` : conversion en nombre ;
- `string( expression )` : conversion en chaîne ;
- `concat( chaîne1, chaîne2 )` : concaténation ;
- `contains( chaîne1, chaîne2 )` : tester si `chaîne1` contient `chaîne2` ;
- `floor( nombre décimal )` : arrondi inférieur (10.9 devient 10, par exemple) ;
- `ceil( nombre décimal )` : arrondi supérieur (10.1 devient 11, par exemple) ;
- `round( nombre décimal )` : arrondi au plus juste (10.4 devient 10 et 10.6 devient 11, par exemple) ;
- `count( NodeSet? )` : nombre de nœuds ;
- `position()` : position courante commençant par 1 ;
- `last( NodeSet? )` : dernière position ;

- `name( NodeSet? )` : nom du nœud (tag s'il s'agit d'unélément) avec préfixe éventuel ;
- `local-name( NodeSet? )` : nom du nœud sans préfixe ;
- `namespace-uri( NodeSet? )` : espace de noms ;
- `generate-id( NodeSet? )` : génération d'un identifiant unique.

Vous retrouverez l'ensemble des fonctions sur le site du W3C à l'adresse : *http://www.w3.org/TR/xpath#corelib*

Quelques exemples :

```
child::personne[3]
```

Renvoie le troisième élément fils `personne`.

```
child::personne[last()]
```

Renvoie le dernier élément `personne`.

```
child::personne[position()>1]
```

Retourne tous les éléments fils `personne`, sauf le premier.

```
count( descendant ::personne )
```

Donne le nombre d'éléments `personne`.

Exercice 4

Quelques expressions XPath au format non abrégé.

Soit le document XML suivant :

```
<livre titre="Mon livre">
<auteurs>
 <auteur nom="nom1" prenom="prenom1"/>
 <auteur nom="nom2" prenom="prenom2"/>
</auteurs>
<sections>
 <section titre="Section1">
  <chapitre titre="Chapitre1">
   <paragraphe>Premier paragraphe</paragraphe>
   <paragraphe>Deuxième paragraphe</paragraphe>
  </chapitre>
 </section>
<section titre="Section2">
  <chapitre titre="Chapitre1">
   <paragraphe>Premier paragraphe</paragraphe>
   <paragraphe>Deuxième paragraphe</paragraphe>
  </chapitre>
 </section>
</sections>
</livre>
```

Écrivez les expressions XPath suivantes au format non abrégé :

– trouver la liste des chapitres de la première section ;

– trouver la liste des attributs du premier auteur ;

– trouver la valeur (fonction `string`) de l'attribut `nom` du deuxième auteur ;

– trouver la liste des chapitres contenant deux paragraphes ;

– trouver la liste des chapitres dont un paragraphe possède le mot `Premier` ;

– trouver la liste des sections ayant un chapitre ;

– trouver la liste des éléments ayant un seul attribut ;

– trouver la liste des éléments ayant un ancêtre `sections`, sous deux formes ;

– trouver la liste des attributs `titre` ;

– trouver la liste des éléments ayant deux fils et pas d'attributs ;

– trouver la liste des sections sans paragraphe ;

– trouver la liste des éléments dont le texte contient le mot `paragraphe`.

La forme abrégée des requêtes XPath

La forme abrégée est plus intuitive, notamment par son analogie avec les chemins (paths) de fichiers. Elle ne couvre pas tous les axes de la forme non abrégée.

Tous d'abord l'axe `child` est implicite.

Si nous écrivons, par exemple :

```
personne/nom
```

nous désignons alors tous les éléments fils `nom` des éléments fils `personne` (selon le nœud de départ).

L'opérateur `//` désigne tous les descendants.

Par exemple :

```
//personne
```

représente tous les éléments `personne` de notre arbre.

L'attribut est préfixé par l'opérateur `@`.

Par exemple :

```
/contacts/personne/@num
```

retourne tous les attributs `num` de tous les éléments `personne` fils de la racine `contacts`.

Le texte est désigné par la fonction `text()`.

Par exemple :

```
/contacts/personne/adresse/text()
```

retourne toutes les adresses sous forme de contenu texte des éléments fils `personne` de la racine `contacts`.

L'opérateur `..` désigne l'élément parent.

Par exemple :

```
//nom[ name( .. ) = 'personne' ]
```

retourne tous les descendants `nom` dont le parent a pour nom `personne`.

Enfin l'opérateur `.` représente le nœud courant.

Par exemple :

```
//personne[./@num=10]
```

retourne tous les descendants `personne` dont l'attribut `num` à pour valeur l'entier 10.

Exercice 5

Quelques expressions XPath au format abrégé.

À l'aide du même document que dans l'exercice 4 :

– trouver la liste de nœuds `auteur` ;

– trouver la liste de tous les nœuds `section` ;

– trouver la liste des chapitres de la première section ;

– trouver la liste des attributs du premier auteur ;

– trouver la valeur de l'attribut `nom` du deuxième auteur ;

– trouver la liste des sections avec deux chapitres ;

– trouver la liste des paragraphes dont le parent a pour titre `Chapitre1`.

La gestion des espaces de noms dans les requêtes XPath

Pour gérer les espaces de noms, il existe plusieurs possibilités :

- Utiliser la fonction `namespace-uri` pour filtrer certains éléments ou attributs.
- Préfixer chaque élément ou attribut. L'association entre le préfixe et l'espace de noms est indépendante du document XML et devra se faire hors de l'expression XPath.

Exemples :

```
entreprise[namespace-uri(.)="http://www.site1.com"]
```

Cette expression filtre les éléments fils `entreprise` dont l'espace de noms est `http://www.site1.com`.

```
site1:entreprise/site1:bureau/site2:machine
```

Dans cette requête, les préfixes `site1` et `site2` sont associés à des espaces de noms distincts.

La version 2.0 de XPath

XPath 2.0 est une véritable refonte et provient d'une fusion entre le language XQuery et XPath 1.0. XPath 2.0 est donc beaucoup plus sophistiqué que XPath 1.0 et ressemble, par certains côtés, au langage SQL.

Voici quelques caractéristiques de ce langage :

- Support des types prédéfinis pour les schémas W3C (19 types comme les dates, URI...).
- Toute réponse est une séquence ordonnée de valeurs (nœud, entier...) et non plus un `NodeSet` sans ordre significatif comme dans XPath 1.0.
- Opérateurs d'intersection, union, différence (`intersect`, `union`, `except`).
- Opérateurs de quantification (`some`, `every`, `satisfies`).
- Autres opérateurs (`for`, `in`, `where`, `sortby`, `let`, `return`).
- Expressions conditionnelles (`if`, `then`, `else`).
- Possibilité de créer de nouvelles fonctions (`define function`).
- Gestion des espaces de noms (déclaration de préfixe par `declare namespace`).
- Test de type (`typeswitch`, `instance of`).

Le document XML suivant est celui que nous utiliserons dans la suite de cet ouvrage pour nos exemples Xpath 2.0 :

```
<liste>
 <element val="11"/>
 <element val="22"/>
 <element val="33"/>
</liste>
```

La gestion par séquence

La séquence est une suite de valeurs. Il faut noter que comme l'ensemble `NodeSet` est automatiquement remplacé par une séquence, les expressions XPath 1.0 restent donc toujours possibles.

On peut, par exemple, créer une séquence avec deux éléments de la façon suivante :

```
( /liste/element[2],/liste/element[1] )
```

L'union est possible via l'opérateur | avec, par exemple :

```
( /liste/element[2],/liste/element[1] ) | (/liste/element[3] )
```

L'intersection de deux séquences peut être réalisée par l'opérateur `intersect`.

Par exemple :

```
( /liste/element[2],/liste/element[1] ) intersect ( /liste/element[2] )
```

La différence entre deux séquences est réalisée par l'opérateur `except`.

Par exemple :

```
( /liste/element[2],/liste/element[1] ) except ( /liste/element[2] )
```

La taille d'une séquence peut être évaluée par la fonction `count`.

Par exemple :

```
count( ( 1, 2 ,3 ) ) : 3
```

La somme d'une séquence s'obtient par la fonction `sum`.

Par exemple :

```
sum( ( 1, 2 ,3 ) ) : 6
```

La moyenne d'une séquence peut être calculée par la fonction `avg`.

Par exemple :

```
avg( //element/@val ) : 22
```

La plus petite valeur d'une séquence s'obtient par la fonction `min`.

```
min( //element/@val ) : 11
```

La plus grande valeur d'une séquence s'obtient par la fonction `max`.

```
max( //element/@val ) : 33
```

La plus petite séquence sans redondance de valeur s'obtient par la fonction `distinct-values`.

Par exemple :

```
distinct-values( (2, 3, 4, 2 ) ) : ( 2, 3, 4 )
```

L'utilisation de variables

Les variables vont nous servir à stocker des valeurs intermédiaires. On déclare une variable par l'opérateur `let`. On préfixe une variable par `$`, comme en PHP.

Voici un exemple :

```
let $a := /liste/element[1]/@val
let $b := /liste/element[2]/@val
return sum( ( number( $a ), number( $b ), 44 ) )
```

La variable `$a` contient la valeur de l'attribut `val` du premier `element`, alors que la variable `$b` contient celle du second `element`. L'opérateur `return` permet ensuite de retourner la somme de ces valeurs avec le nombre 44 (résultat 77).

La variable peut également contenir une partie de l'arbre XML :

```
let $a := /liste/element[2]
return  $a
```

Dans cet exemple, la variable `$a` contient le deuxième nœud `element`.

L'intégration de boucles

XPath 2.0 supporte les boucles `for` que l'on retrouve à peu près dans tous les langages procéduraux. Cette boucle va servir à parcourir chaque élément d'une séquence et retourner un nouveau résultat, chaque élément étant stocké dans une variable employable dans le corps de la boucle. À chaque passage dans la boucle, l'instruction `return` renvoie une partie de la réponse.

Par exemple :

```
for $i in //element
return <item val="{$i/@val}"></item>
```

Pour chaque balise `element` de notre document XML, une balise `item` va être produite avec un attribut `val` dont la valeur sera celle de l'attribut `val` du document XML. À noter que les accolades désignent le résultat de l'expression.

Avec cet autre exemple :

```
for $ii in //element
return ($ii/@val(), " ")
```

nous créons une liste de tous les attributs `val` (séparation par un blanc).

On peut également passer une boucle en argument d'une fonction (puisque la boucle retourne une séquence), par exemple, de cette manière :

```
sum( for $i in //element
return  $i/@val )
```

Les boucles de type FLWOR

Le terme FLWOR (à prononcer *flower*) est une abréviation pour `For Let Where Order` et `Return`.

Voici quelques exemples d'usage :

```
let $m := avg( //element/@val )
for $i in //element where ( $i/@val > $m  )
return  $i
```

Ces instructions permettent d'obtenir les éléments dont la valeur portée par l'attribut `val` est supérieure à la moyenne.

```
let $m := avg( //element/@val )
for $i in ( //element, <element val="23"/> ) where ( $i/@val > $m  ) order by $i/@val
return  $i
```

Dans cet autre exemple, nous effectuons un tri par ordre croissant de la séquence résultat (ordre lexicographique). Nous avons également ajouté une balise supplémentaire avec la valeur `23` pour l'attribut `val`.

Il est possible d'effectuer plusieurs boucles `for` imbriquées pour des opérations de jointure.

Exemple sur le document XML suivant :

```
<liste>
<element val="11"/>
<element val="22"/>
<element val="33"/>
<test val="12"/>
<test val="23"/>
<test val="34"/>
</liste>
```

Avec la requête XPath 2.0 suivante :

```
for $i in //element
for $j in //test where $j/@val > 2 * $i/@val
return $j
```

nous avons filtré tous les éléments `test` dont la valeur de l'attribut `val` est supérieure au double de la valeur de l'attribut `val` de chaque balise `element`.

Le contrôle de flux

On réalise des conditions avec les mots-clés `if`, `then` et `else`. À noter que `else` est obligatoire, car il faut dans tous les cas fournir une réponse au moteur d'évaluation (à ne pas confondre avec une séquence vide).

Voici un exemple :

```
(: Résultat : "ok" :)
if ( /liste/element )
then "ok" else "erreur"
```

Dans cet exemple, nous retournons le message `ok` si la racine `liste` contient au moins un fils `element` et `erreur` dans le cas contraire. À noter la présence de commentaires entre `(:` et `:)`.

Les opérateurs `some` et `every` servent à vérifier des conditions et retournent donc une valeur booléenne. L'opérateur `some` teste simplement si la condition est vérifiée pour au moins un élément d'une séquence donnée, alors que l'opérateur `any` vérifie que cette condition est vérifiée pour tous les éléments de la séquence.

Voici deux exemples :

```
(: Résultat : true :)
some $e in liste/element satisfies $e/@val=11
```

Ici, nous testons s'il existe au moins un élément dont l'attribut `val` à pour valeur `11`.

```
(: Résultat : true :)
every $e in liste/element satisfies ( $e/@val>=11 and $e/@val<=33 )
```

Dans cet autre cas, il faut que l'attribut `val` de tous les éléments ait une valeur comprise entre `11` et `33` pour que le résultat de l'expression soit vrai.

Le casting des valeurs

Le *casting* est une opération qui consiste à associer un type à une valeur, chaque valeur n'existant que dans le contexte d'un type. Pour le texte, XPath 2.0 s'appuie sur les types simples prédéfinis dans les schémas W3C.

L'instruction `instance of` vérifie la conformité d'une valeur avec un type et renvoie une valeur booléenne.

Par exemple, cette expression retourne la valeur vrai :

```
11 instance of xs:integer
```

L'instruction `cast as` sert à convertir une valeur dans un autre type. Cela n'a bien sûr de sens que si le type recherché peut héberger une telle valeur.

Par exemple :

```
2 cast as xs:boolean
```

Ici la valeur retournée sera `true`, car une valeur positive sera toujours associée à vrai alors que la valeur 0 sera associée à faux.

Cet autre exemple ne sera pas possible :

```
2 cast as xs:date
```

pour la raison simple que le moteur d'exécution n'a aucun moyen de faire le lien entre le chiffre `2` et une date.

Pour éviter de faire des casts incorrects, l'instruction `castable as` indique si un cast est possible.

L'expression `2 castable as xs:date` renverra alors la valeur `false`.

On peut réaliser des tests plus sophistiqués, comme :

```
let $v:= //element[ 1 ]/@val
return if ( $v castable as xs:date ) then
$v else
$v cast as xs:string
```

Ces instructions retournent une séquence avec une valeur dont le type peut être une date ou une chaîne.

Des fonctions portant le nom de chaque type servent également à construire une valeur directement typée.

Par exemple :

```
xs:date("2007-06-21")
```

La valeur retournée sera typée comme étant une date.

Les fonctions dans XPath 2.0

Les fonctions que nous avions dans XPath 1.0 sont également disponibles dans XPath 2.0. L'utilisateur de XQuery ou de XSLT dispose de la possibilité de créer ses propres fonctions. Pour éviter les collisions entre les différents noms de fonctions, différents espaces de noms sont disponibles :

- L'espace de noms `http://www.w3.org/2001/XMLSchema` contient les fonctions de construction et est associé au préfixe `xs`.
- L'espace de noms `http://www.w3.org/2005/xpath-functions` contient toutes les fonctions XPath (y compris celles de la version 1.0) et est associé au préfixe `fn`.

L'un des points intéressants par rapport à XPath 1.0 est le support des expressions régulières pour le traitement des chaînes. Si vous n'êtes pas familiarisé avec les expressions régulières, sachez qu'il s'agit d'un concept visant à exprimer une séquence de caractères vérifiant une certaine structure. Cela donne donc une forme lexicale qui désigne un ensemble de valeurs possibles et permet de généraliser certains traitements.

La syntaxe des expressions régulières est là même que pour les schémas (voir le chapitre 3) : *http://www.w3.org/TR/xmlschema-2/#regexs*.

Voici juste quelques rappels :

- `^` : désigne le début d'un motif.
- `$` : désigne la fin d'un motif.
- `.` : désigne n'importe quel caractère (pour avoir explicitement le point, il faut le préfixer par un slash).
- `[ … ]` : correspond à un intervalle de valeurs, par exemple `[0-9]` pour tous les caractères chiffres.
- `{ .. }` : sert à définir des quantités. Si une valeur entière est précisée, alors cela signifie exactement `n` fois ; si deux valeurs entières `m` et `n` sont indiquées, cela signifie entre `m` et `n` fois. Par exemple, `z{2,10}` signifie un caractère `z` présent de deux à dix fois. Autre exemple `z{2,}` signifie au moins 2 fois.
- Des opérateurs de cardinalités tels `?`, `+`, `*` désignent, respectivement, une valeur optionnelle, une valeur présente au moins une fois et une valeur absente ou présente n fois.
- (…) : les parenthèses servent à stocker des valeurs dans des pseudo-variables ($1, $2…). Le numéro de la variable est croissant selon l'ordre d'apparition des parenthèses, de la gauche vers la droite. Cela sert essentiellement lors d'opérations de remplacement où l'on souhaite remettre dans le résultat une partie du texte.
- `\d`, `\w`, `\s` désignent respectivement un nombre, un caractère de mot et un blanc. La même chose en majuscule caractérise le contraire.

Nous avons trois fonctions :

- `fn:matches` : elle vérifie si la chaîne en premier argument correspond à l'expression régulière en deuxième argument. Un troisième argument peut servir à passer des paramètres (*flags*), comme `s` pour que le point gère les retours à la ligne, ou bien `i` pour ignorer la casse.
- `fn:replace` : retourne la chaîne résultant du remplacement, dans la chaîne passée en premier argument, de toutes les occurrences exprimées par l'expression régulière fournie en deuxième argument par la valeur donnée en troisième argument.
- `fn:tokenize` : retourne une séquence de chaînes (`xs :string`) en découpant la chaîne passée en premier argument grâce au séparateur donné en deuxième argument, sous la forme d'une expression régulière.

Quelques exemples :

```
fn:matches( "ABBBC", "AB+C") : Répond vrai
fn:replace( "01-02-03", "(\d{2})-(\d{2})-(\d{2})", "$3/$2/$1") : Donne 03/02/01.
fn:tokenize( "01-02-03", "-") : Retourne la séquence 01 02 03
```

Exercice 6

Quelques expressions XPath 2.0

À partir du document de l'exercice 4, que vous modifierez pour avoir de un à trois paragraphes dans les chapitres, réalisez les expressions suivantes :

– afficher la liste des chapitres avec plus de deux paragraphes à l'aide d'une expression FLWOR ;

– afficher une séquence de texte avec la liste des sections et les chapitres liés.

Le format XSLT

XSLT (*Extensible Stylesheet Language Transformations*) est un langage XML qui sert à passer d'un format XML à un autre format texte (XML, XHTML/HTML, CSV…).

Il existe deux recommandations du W3C respectivement pour des versions 1.0 (*http://www.w3.org/TR/xslt/*) et 2.0 (*http://www.w3.org/TR/xslt20/*). La version 1.0 reste davantage exploitée, au moment de la rédaction de cet ouvrage, que la version 2.0 pour plusieurs raisons :

- poids de l'existant ;
- les navigateurs ne gèrent que la version 1.0 ;
- encore trop peu de librairies pour la version 2.0.

XSLT 1.0 est fortement couplé à XPath 1.0 alors que XSLT 2.0 est associé à XPath 2.0.

L'algorithme de transformation

On pourrait résumer l'algorithme des transformations XSLT par le principe suivant. L'arbre est parcouru grâce à des expressions XPath ; en fonction de la localisation dans l'arbre, certains blocs d'instructions sont exécutés. Le passage dans une partie de l'arbre est représenté par un ensemble de nœuds, résultat d'une expression XPath, que l'on qualifie de nœuds courants. Ces nœuds deviendront les nœuds de référence pour les expressions XPath relatives qui suivront et ainsi de suite.

Un document XSLT est l'agglomération de plusieurs langages. D'une part, les instructions propres à XSLT (y compris XPath) et d'autre part, le document émis en résultat qui peut être un autre langage XML… La distinction entre les différents langages se fait grâce aux espaces de noms, les éléments XSLT étant dans l'espace de noms `http://www.w3.org/1999/XSL/Transform`.

Le langage XSLT 1.0

Nous allons maintenant aborder les principaux éléments du langage XSLT 1.0. On a l'habitude de les préfixer par `xsl`.

Squelette d'un document XSLT 1.0

Voici le squelette d'un document XSLT 1.0 :

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

L'instruction `output`, qui se place sous la racine `stylesheet`, désigne la nature du document en sortie. Voici quelques attributs :

- `method` = `text`, `html`, `xml` ;
- `encoding` = jeu de caractères ;
- `indent` = `yes` ou `no` ;
- `media-type` = type MIME de sortie.

Point principal : les éléments template

Le `template` est une instruction incontournable. Elle représente un bloc d'instructions qui seront exécutées pour certains nœuds courants. Un `template` joue un peu le rôle de méthode dans un langage procédural, tel que le C ou le Pascal. On ne peut donc pas imbriquer des templates qui sont indépendants et sont appelés, soit directement soit indirectement, selon les résultats de l'instruction `apply-templates`.

Lorsqu'aucun `template` ne peut traiter un nœud, des templates prédéfinis offrent un comportement par défaut :

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

```
</xsl:template>
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="processing-instruction()|comment()"/>
```

Ce comportement consiste à parcourir tous les éléments fils et à afficher les textes ou les attributs. Par contre, les instructions de traitement et les commentaires sont ignorés.

Les principaux attributs de l'instruction `template` sont :

- `name` : un nom, si on appelle le `template` directement (indépendamment des nœuds courants).
- `match` : il s'agit d'un chemin de localisation XPath qui indique pour quels nœuds le `template` peut être appelé.
- `mode` : le mode sert à donner un contexte d'usage au `template`, l'idée étant que pour des mêmes nœuds courants, on puisse invoquer plusieurs templates.
- `priority` : il s'agit d'un nombre décimal qui sert à aider le moteur de transformation à choisir entre différents templates acceptant les mêmes nœuds.

Lorsqu'une transformation XSLT commence, le nœud courant est d'office le document XML. Ce nœud courant est désigné par l'instruction XPath `/`.

Le squelette d'un document XSLT effectuant une transformation vers un document XHTML/HTML serait donc, par exemple, sous cette forme :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
<body>...</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Dans cet exemple, le template *principal* (un peu à la manière d'une méthode `main` en C/C++ ou Java) va générer les balises englobantes propres à XHTML/HTML. Le contenu de la balise `body` sera dépendant d'autres instructions XSLT.

Voici un autre exemple qui vous montre l'usage de l'attribut `match` :

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

Nous avons ici créé deux templates : le premier sera invoqué pour chaque nœud courant correspondant à l'élément `personne`, alors que le second sera appelé pour l'élément `email`.

Pour déclencher l'appel d'un template avec changement des nœuds courants, on s'appuie sur l'instruction `apply-templates`. Sans attribut `select`, cette instruction parcourt tous les nœuds fils (y compris les nœuds texte). Dans le cas contraire, le résultat de l'expression XPath de cet attribut sert à invoquer un template (ou celui par défaut, si aucun template ne correspond). L'attribut `mode` peut être ajouté pour n'activer qu'un template en particulier (celui avec le même attribut et la même valeur).

Voici un extrait XSLT :

```
<xsl:template match="/">
 <xsl:apply-templates select="contacts/personne"/>
</xsl:template>

<xsl:template match="personne">
...
</xsl:template>
```

Dans cet extrait, le premier `template` est invoqué d'office. Une requête XPath est alors effectuée pour que les nœuds courants soient des éléments `personne` (fils de la racine `contacts`), ce qui déclenche, pour chacun de ces nœuds, le dernier `template`.

Les paramètres des templates

Des paramètres peuvent être ajoutés soit sous la racine XSLT, soit au niveau d'un template. Dans le premier cas, les valeurs de ces paramètres seront renseignées avant le processus de transformation (via une application). Dans le deuxième cas, les valeurs seront définies lors de l'appel au template.

Voici un exemple :

```
<xsl:template match="/">
 <xsl:apply-templates select="contacts/personne">
  <xsl:with-param name="P1">10</xsl:with-param>
  <xsl:with-param name="P2">20</xsl:with-param>
 </xsl:apply-templates>
</xsl:template>

<xsl:template match="personne">
 <xsl:param name="P1"/>
 <xsl:param name="P2"/>
 <xsl:value-of select="$P1+$P2"/>
 ...
</xsl:template>
```

Nous invoquons le dernier `template` comme auparavant, mais en spécifiant les deux paramètres `P1` et `P2` avec pour valeurs respectives `10` et `20`. Ces paramètres sont ensuite récupérés grâce à l'instruction `param`. La dernière instruction `value-of` a pour rôle de produire le résultat de l'expression XPath sous forme de texte, chaque paramètre étant considéré comme une variable XPath préfixée par `$`.

L'appel de template explicite

L'appel à un template indépendamment des nœuds courants est également possible via l'instruction `call-template`. Cette dernière s'appuie sur l'attribut `name` et sa valeur, que l'on doit également retrouver dans le template invoqué.

Voici un exemple :

```
<xsl:template ...>
 <xsl:call-template name="ACTION">
  <xsl:with-param name="MESSAGE">Hello world</xsl:with-param>
 </xsl:call-template>
...
</xsl:template>
<xsl:template name="ACTION">
 <xsl:param name="MESSAGE"/>
 <b><xsl:value-of select="'$MESSAGE'"/></b>
</xsl:template>
```

Le template nommé `ACTION` est invoqué dans le premier template. Cela n'entraîne aucun changement de nœuds courants. Dans cet exemple, nous avons également intégré le paramètre `MESSAGE`.

Les boucles

Une boucle joue un peu le même rôle que l'instruction `apply-templates` mais à un niveau plus local, puisque cette instruction ne déclenche pas de déplacement vers un autre `template`. L'attribut `select` et son expression XPath associée vont servir à construire un NodeSet. Pour chaque nœud de ce résultat, les instructions de la boucle vont être exécutées. Le nœud courant va également varier à chaque itération. Bien entendu, en sortie de boucle, nous retrouvons le même contexte qu'en entrée.

Voici un exemple :

```
<xsl:for-each select="personne">
<xsl:value-of select="@nom"/>
</xsl:for-each>
```

Ici, tous les éléments fils `personne` sont parcourus et, pour chacun d'eux, la valeur de l'attribut `nom` est produite en résultat.

Les tris sur les boucles ou les appels de template

Lorsque des instructions (template ou boucle) s'appliquent à un ensemble de nœuds (NodeSet), il est possible de trier cet ensemble grâce à l'instruction `sort`. Pour chaque instruction `sort` présente, on spécifie sur quel critère doit s'effectuer le tri grâce à l'attribut `select`, qui contient une expression XPath relative au nœud courant.

Cette instruction contient les attributs suivant :

- `select` : une expression XPath désignant une zone dans le nœud courant sur laquelle doit s'effectuer le tri ;
- `data-type` : type de données, `text` ou `number` ;

- `order` : ordre du tri, `ascending` ou `descending` ;
- `case-order` : priorité pour les majuscules/minuscules, `upper-first` ou `lower-first`.

Voici un exemple avec le document XML suivant :

```
<carnet>
 <personne nom="dupont" prenom="jean"/>
 <personne nom="dupont" prenom="arthur"/>
 <personne nom="dupond" prenom="martin"/>
</carnet>
```

Le code ci-après effectue un tri de toutes les personnes en fonction de leur `nom` et de leur `prenom`. Le tri est fait dans l'ordre d'exécution des instructions `sort`, qui sont alors placées de la priorité la plus forte à la priorité la plus faible.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<xsl:for-each select="//personne">
  <xsl:sort select="@nom"/>
  <xsl:sort select="@prenom"/>
  <xsl:value-of select="concat(@nom,' ',@prenom)"/>
 </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

En résultat, nous obtenons `dupond martin`, `dupont arthur`, `dupont jean`, le tri ayant eu lieu en priorité sur le `nom`, puis sur le `prenom`.

Le contrôle de flux

Le contrôle de flux consiste à effectuer des actions conditionnellement à la réussite de certains tests.

Le premier cas est l'instruction `if` avec l'attribut `test`. Un exemple d'usage devrait suffire à comprendre son fonctionnement :

```
<xsl:for-each select="//personne">
 <xsl:if test="./@nom='dupond'">
 Bienvenue Mr Dupond
 </xsl:if>
 <xsl:if test="not( ./@nom='dupond')">
 Bienvenue Mme <xsl:value-of select="@nom"/>
 </xsl:if>
</xsl:for-each>
```

Remarque

Il n'existe pas d'instruction `else` en XSLT. C'est à vous d'écrire une autre instruction `if` avec la négation de l'instruction précédente, ce qui n'est ni pratique ni performant. L'utilisation d'une variable améliore les choses ; vous pouvez également passer à XSLT 2.0 (et XPath 2.0).

Lorsque des tests à répétition sont nécessaires, les instructions `choose`, `when` et `otherwise` sont d'un emploi plus pratique. On retrouve ce type d'instructions dans la plupart des langages de programmation (`switch/case`, `select/case`…).

Voici un exemple de fonctionnement :

```
<xsl:choose>
 <xsl:when test="@nom='dupond'">
 Bienvenue Mr Dupond
 </xsl:when>
 <xsl:when test="@nom='dupont'">
 Bienvenue Mme Dupont
 </xsl:when>
 <xsl:otherwise>
 Bienvenue
 </xsl:otherwise>
</xsl:choose>
```

Le code encadré par l'instruction `when` n'est appelé que si le test lié à l'attribut `test` réussit ; tous les autres cas sont alors ignorés. Si aucun test ne réussit, le code encadré par l'instruction `otherwise` est alors exécuté.

Les nœuds texte

Pour produire du texte, la première instruction, et probablement la plus courante, est l'instruction `value-of`. Elle produit un texte qui est le résultat de l'expression XPath passée via l'attribut `select`. Si l'attribut `disable-output-escaping` prend la valeur `yes` alors certains caractères seront laissés tel quel (comme les caractères `<` ou `&`).

Voici un exemple d'utilisation :

```
<xsl:template match="/">
 <xsl:for-each select="//personne">
  <h1><xsl:value-of select="@nom"/>
  <xsl:value-of select="@prenom"/></h1>
 </xsl:for-each>
</xsl:template>
```

Ce code permet d'afficher chaque attribut `nom` et `prenom` pour les éléments `personne` de notre document XML source.

On peut créer explicitement des nœuds texte via l'instruction `text`. Cette dernière peut également garder certains caractères en utilisant l'attribut `disable-output-escaping`. C'est souvent pratique pour créer une séparation entre plusieurs valeurs.

La numérotation en sortie

Un numéro peut être automatiquement géré pour un nœud. On peut prendre l'exemple d'une numérotation de chapitre dans une structure représentant un article ou un livre.

Le premier cas, le plus simple, est l'usage de l'attribut `value`. L'expression XPath passée en argument est évaluée et le résultat est converti en nombre (grâce à la fonction `number`) avant de passer dans le document final.

La numérotation peut être typée via l'attribut `format`. Cet attribut possède une valeur qui joue un peu le rôle d'un masque pour le nombre. Le choix des chiffres ou des lettres aiguille le moteur de transformation à formater correctement le nombre (par exemple `1.`, `a`, `A.1`, `I`…).

Lorsqu'on ne précise rien, l'instruction `number` retourne la position du nœud courant - cette position étant liée à la valeur de l'attribut `level`. Pour `single`, il s'agit de son numéro par rapport à son parent, pour `multiple` la numérotation peut prendre en compte plusieurs ancêtres et pour `any` il s'agit d'un numéro unique selon le parcours de l'arbre.

Lorsque le numéro est composé de plusieurs chiffres, l'attribut `count` désigne les nœuds qui doivent intervenir dans le calcul du chiffre (par exemple `A.1`, `A.2`, `B.1`, `B.2`…).

Voici un exemple :

```
<carnet>
<personne nom="dupont">
 <poste>Plongeur</poste>
 <poste>Cuistot</poste>
</personne>
<personne nom="dupont">
 <poste>Pompier</poste>
 <poste>Astronaute</poste>
</personne>
</carnet>
```

Soit la feuille de styles suivante :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/">
  <xsl:for-each select="//poste">
   <xsl:number count="poste|personne" level="multiple" format="A.I"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="."/>
   <xsl:text> </xsl:text>
 </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Ce code produit alors le texte suivant : `A.I Plongeur A.II Cuistot B.I Pompier B.II Astronaute`. En effet, il permet de compter les éléments `poste` et `personne` et utilise une numération multiple (attribut `level`).

Déclaration et utilisation de variable

En XSLT, une variable porte mal son nom puisque sa valeur initiale ne peut pas être changée. Elle sert à stocker un résultat intermédiaire.

Un exemple :

```
<xsl:for-each select="//poste">
 <xsl:variable name="type" select="."/>
 <xsl:if test="contains($type, 'Astro')">Métier sur les étoiles</xsl:if>
</xsl:for-each>
```

On peut également stocker dans une variable une partie de l'arbre (un fragment).

Un autre exemple :

```
<xsl:for-each select="//personne">
 <xsl:variable name="p" select="."/>
 <xsl:value-of select="$p/@nom"/>
</xsl:for-each>
```

La génération d'un nouvel arbre en sortie

Il existe plusieurs méthodes pour produire un nouveau document XML. Commençons par la plus simple qui consiste à positionner des balises comme résultat.

```
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
 <xsl:for-each select="//personne">
  <unePersonne></unePersonne>
 </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

On génère donc autant d'éléments `unePersonne` qu'il y a d'éléments `personne`. L'un des problèmes que l'on rencontre concerne les attributs : on ne peut pas insérer une instruction `value-of` dans un attribut, la valeur d'un attribut ne pouvant contenir d'éléments. Heureusement, il existe une solution avec des accolades qui indiquent au moteur de transformation de faire une évaluation XPath dans une portion d'un texte.

Voici un exemple :

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

<xsl:output method="xml"/>
 <xsl:template match="/">
 <xsl:for-each select="//personne">
  <unePersonne nomPrenom="{@nom}"></unePersonne>
 </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Nous avons obtenu en résultat :

```
<unePersonne nomPrenom="dupont"/><unePersonne nomPrenom="dupont"/>
```

L'instruction `attribute` peut jouer le même rôle de cette manière :

```
<xsl:for-each select="//personne">
<unePersonne>
<xsl:attribute name="nomPrenom"><xsl:value-of select="@nom"/></xsl:attribute>
</unePersonne>
</xsl:for-each>
```

Un nœud attribut est donc créé et associé à son parent. Le contenu de cette instruction correspond à la valeur de l'attribut.

L'instruction `element` sert à créer un nœud élément. L'attribut `namespace` permet de lui associer un espace de noms. L'attribut `use-attribute-sets` sert à lui associer un groupe d'attributs.

Voici un exemple produisant un élément `unePersonne` :

```
<xsl:element name="unePersonne">
<xsl:attribute name="nomPrenom"><xsl:value-of select="@nom"/></xsl:attribute>
</xsl:element>
```

Cet autre exemple utilise un groupe d'attributs grâce à l'élément `attribute-set` :

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

<xsl:attribute-set name="attDef">
 <xsl:attribute name="id">10</xsl:attribute>
 <xsl:attribute name="def"><xsl:value-of select="position()"/></xsl:attribute>
</xsl:attribute-set>

<xsl:template match="/">
 <xsl:for-each select="//personne">
  <xsl:element name="unePersonne" use-attribute-sets="attDef">
  <xsl:attribute name="nomPrenom"><xsl:value-of select="@nom"/></xsl:attribute>
  </xsl:element>
 </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Le groupe d'attributs est désigné par le nom `attDef`.

Il est également possible de réaliser une copie de nœud en sortie. Tout d'abord, l'instruction `copy` réalise une copie du nœud courant et de l'espace de noms. Cependant, dans le cas des éléments, les attributs et contenus ne sont pas copiés automatiquement.

Voici un exemple :

```
<xsl:for-each select="//personne">
 <xsl:copy>
  <xsl:attribute name="nomPrenom"><xsl:value-of select="@nom"/></xsl:attribute>
 </xsl:copy>
</xsl:for-each>
```

Cela produit le document suivant :

```
<personne nomPrenom="dupont"/><personne nomPrenom="dupont"/>
```

Si nous souhaitons copier le contenu et les attributs, il faut alors réaliser un `template` dans ce style :

```
<xsl:template match="/">
<xsl:for-each select="//personne">
 <xsl:copy>
  <xsl:apply-templates select="node()" mode="copie"/>
 </xsl:copy>
</xsl:for-each>
</xsl:template>
<xsl:template match="personne//*|personne/@*" mode="copie">
 <xsl:copy>
  <xsl:apply-templates mode="copie"/>
 </xsl:copy>
</xsl:template>
```

Le deuxième `template` effectue une copie en profondeur. Le `mode` garantit qu'aucun autre `template` ne sera appelé et que seule la copie va être effectuée.

L'instruction `copy-of` est capable d'effectuer la copie complète de l'arbre. L'attribut `select` sert à désigner les nœuds que l'on souhaite copier.

Voici un exemple :

```
<xsl:stylesheet
version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
 <carnetBis>
  <xsl:copy-of select="//personne"/>
 </carnetBis>
</xsl:template>
</xsl:stylesheet>
```

Ce qui va produire le texte suivant :

```
<carnetBis>
<personne nom="dupont">
 <poste>Plongeur</poste>
 <poste>Cuistot</poste>
</personne>
<personne nom="dupont">
 <poste>Pompier</poste>
 <poste>Astronaute</poste>
</personne>
</carnetBis>
```

L'inclusion d'un document XSLT

L'inclusion d'une autre feuille de styles est effectuée par l'instruction `include`, que l'on positionne sous la racine. Cette instruction est assez passive et effectue un remplacement avec le contenu de la feuille de styles importée. On peut se constituer cependant une librairie de feuilles de styles avec les opérations les plus répétées.

Par exemple, voici la feuille de styles `intro.xsl` que l'on souhaite inclure :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template name="intro">
<h1>Hello World
</h1>
</xsl:template>
</xsl:stylesheet>
```

L'inclusion est alors réalisée dans le document suivant :

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

<xsl:output method="html"/>
<xsl:include href="intro.xsl"/>
 <xsl:template match="/">

  <xsl:call-template name="intro"></xsl:call-template>
 </xsl:template>
</xsl:stylesheet>
```

L'attribut `href` fait la liaison avec notre autre feuille de styles ; à noter que nous effectuons ici un accès relatif.

L'importation d'un document XSLT

L'importation est proche de l'inclusion mais agit de manière plus subtile sur les templates présents. L'idée étant que les éléments importés ont toujours une priorité plus faible que les éléments courants. Cela permet de créer, par exemple, une espèce de surcharge (au sens objet avec les méthodes de classes parentes) de telle ou telle partie.

Soit la feuille de styles `intro.xsl` : comme vous pouvez le constater la règle `intro` appelle la règle `introTexte`.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template name="intro">
<h1><xsl:call-template name="introTexte"></xsl:call-template>
</h1>
</xsl:template>
<xsl:template name="introTexte">
Hello World
</xsl:template>
</xsl:stylesheet>
```

Le code suivant permet d'effectuer une importation :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="intro.xsl"/>
<xsl:output method="html"/>
 <xsl:template match="/">
  <xsl:call-template name="intro"></xsl:call-template>
 </xsl:template>
 <xsl:template name="introTexte">Hello World 2</xsl:template>
</xsl:stylesheet>
```

Comme l'inclusion, l'importation utilise l'attribut `href`. Ce qui est important, c'est le fait que nous ayons réécrit la règle `introTexte` qui était appelée par le premier `template` importé. Elle a donc maintenant priorité et lorsque la règle `intro` est appelée, la nouvelle règle `introTexte` l'est également.

Nous avons donc en résultat :

```
<h1>Hello World 2</h1>
```

Exercice 7

Génération d'une page HTML

Effectuez une transformation XSL sur le document de l'exercice 4 pour afficher, dans un document HTML, une table des matières avec les sections et les chapitres. Dans un deuxième temps, complétez le document pour avoir une numérotation des titres et un lien interne vers le détail de chaque partie.

Exercice 8

Génération d'un nouveau document XML

À partir du document de l'exercice 4, utilisez XSL pour réaliser un document XML selon la structure suivante :

```
livre
 titre
 auteurs
  auteur / attribut nomPrenom
 section
  titre
  chapitre
   titre
   para
```

La gestion des espaces de noms dans un document XSLT

La gestion des espaces de noms passe par la déclaration de préfixes (de préférence sur la racine) pour traiter tels éléments ou attributs.

Voici un exemple de document XML :

```
<carnet xmlns="http://www.a.com">
<personne xmlns="http://www.b.com" nom="Dupont">
</personne>
<personne xmlns="http://www.b.com" nom="Dupond">
</personne>
</carnet>
```

Il comporte deux espaces de noms : `http://www.a.com` et `http://www.b.com`. Voici un exemple de feuille de styles pour transformer ce document en un autre document XML :

```
<xsl:stylesheet version="1.0" xmlns:a="http://www.a.com" xmlns:b=
➥"http://www.b.com" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/a:carnet">
<agenda>
  <xsl:for-each select="b:personne">
   <employe>
    <xsl:value-of select="@nom"/>
   </employe>
  </xsl:for-each>
 </agenda>
</xsl:template>
</xsl:stylesheet>
```

Comme vous pouvez le constater, il suffit de préfixer les éléments des requêtes XPath pour que le moteur puisse filtrer les bons éléments. Nous obtenons le document résultat suivant :

```
<agenda
xmlns:b="http://www.b.com" xmlns:a="http://www.a.com">
<employe>Dupont</employe>
<employe>Dupond</employe>
</agenda>
```

Le moteur de transformation a ajouté les déclarations de préfixes car rien n'indique si le document généré est un document XML final ou s'il doit faire l'objet d'autres requêtes XPath.

Pour empêcher cet ajout dans le document final, il suffit d'écrire l'attribut `exclude-result-prefixes` sur la racine, avec pour valeur associée `"a b"`, ce qui a pour effet d'ignorer les deux préfixes `a` et `b`. Nous obtenons alors bien le document :

```
<agenda>
<employe>Dupont</employe>
<employe>Dupond</employe>
</agenda>
```

Les extensions de la version 1.0

On trouve quelques moteurs de transformation offrant des extensions, notamment en termes de fonctions.

Xalan (*http://xalan.apache.org/*) offre la possibilité d'invoquer des fonctions Java ou de créer de nouvelles fonctions en JavaScript.

Voici un exemple :

```
<xsl:stylesheet
version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:math="xalan://java.lang.Math"
xmlns:date="xalan://java.util.Date"
>
<xsl:output method="html"/>

<xsl:template match="/">
<xsl:value-of select="math:max(30,20)" />
<xsl:variable name="date" select="date:new()"></xsl:variable>
Nous sommes le <xsl:value-of select="date:getDay($date)"/>
➥/<xsl:value-of select="date:getMonth($date)+1"/>
</xsl:template>

</xsl:stylesheet>
```

Dans cet exemple, nous avons invoqué la méthode statique `max` de la classe `java.lang.Math`. Puis, nous avons créé un objet qui est stocké dans la variable `date`. Nous avons ensuite invoqué des méthodes sur ces objets et avons obtenu la sortie suivante :

```
30
Nous sommes le 4/6
```

Le projet EXSLT (*http://www.exslt.org/*) propose de nombreuses fonctions supplémentaires (toutes écrites en JavaScript).

L'idée est d'importer un ensemble de feuilles de styles selon les fonctions souhaitées.

```
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:math="http://exslt.org/math"
          extension-element-prefixes="math">
<xsl:import href="math.min.template.xsl" />
...
</xsl:stylesheet>
```

Dans l'exemple ci-avant, la fonction qui permet d'obtenir le minimum d'un ensemble de valeurs est importée. Il reste alors à appeler un `template` lié à chaque fonction selon ce genre d'instruction :

```
<xsl:call-template name="math:min">...
</xsl:call-template>
```

Le langage XSLT 2.0

XSLT 2.0 s'appuie sur XPath 2.0, ce qui constitue déjà un changement important.

La création d'une séquence

Comme nous l'avons vu avec XPath 2.0, un résultat est une séquence de valeurs. Dans XSLT 2.0, il est possible de créer une séquence en une ou plusieurs fois grâce à l'instruction `sequence`. L'attribut `select` sert alors à définir le contenu de cette séquence.

Voici un exemple :

```
<xsl:variable name="seq1" as="xs:double+">
<xsl:sequence select="(1,2)"></xsl:sequence>
 <xsl:sequence select="(3)"></xsl:sequence>
</xsl:variable>
<xsl:value-of select="sum($seq1)"></xsl:value-of>
```

L'attribut `as` de la variable sert à typer les éléments de la séquence.

La génération de plusieurs documents en sortie

L'instruction `result-document` sert à créer plusieurs documents en sortie.

Nous allons réutiliser de document XML suivant :

```
<carnet>
<personne nom="dupont">
</personne>
<personne nom="dupond">
</personne>
</carnet>
```

Un fichier contenant le nom de chaque personne, précédé d'un message, peut être produit grâce à la feuille de styles suivante :

```
<xsl:stylesheet
version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
<xsl:template match="/">
<xsl:for-each select="//personne">
<xsl:result-document href="{@nom}.txt" method="text">
Bonjour Mr <xsl:value-of select="@nom"/>
</xsl:result-document>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

L'utilisation de plusieurs documents en entrée

Plusieurs entrées peuvent être parcourues avec la fonction `document`, qui prend en argument une URI vers un fichier XML.

Soient les deux fichiers XML suivants :

carnet.xml

```
<carnet>
<personne nom="dupont">
</personne>
<personne nom="dupond">
</personne>
</carnet>
```

info.xml

```
<carnet>
<info nom="dupont">
 <age>25</age>
</info>
<info nom="dupond">
 <age>26</age>
</info>
</carnet>
```

Si nous voulons travailler sur ces deux documents, nous pouvons écrire :

```
<xsl:stylesheet
version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<xsl:for-each select="document('carnet.xml')//personne">
<xsl:variable name="p" select="@nom"/>
L'age de Mr <xsl:value-of select="$p"/> est :
<xsl:value-of select="document('info.xml')//info[@nom=$p]/age"/>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Comme vous pouvez le constater, la variable `p` est associée au nom de la personne courante ; on recherche alors, dans le fichier `info.xml`, l'âge correspondant.

La fonction `unparsed-text` sert à stocker le contenu d'un fichier sous forme de chaîne.

Une nouvelle forme de boucle : les groupes

XSLT 2.0 introduit une nouvelle forme de boucle : le groupement des valeurs. L'idée est de ne pas parcourir des valeurs mais des groupes de valeurs. Par exemple, si plusieurs éléments peuvent être regroupés en catégories, on souhaite pouvoir faire un inventaire de chaque catégorie.

Voici un exemple :

```
<villes>
 <ville nom="milan"  pays="italy"   pop="5"/>
 <ville nom="paris"  pays="france"  pop="7"/>
```

```
 <ville nom="munich" pays="germany" pop="4"/>
 <ville nom="lyon"   pays="france"  pop="2"/>
 <ville nom="venice" pays="italy"   pop="1"/>
</villes>
```

L'idée est d'arriver à afficher les villes et populations pour chaque pays. En XSLT 1.0, il nous faudrait faire deux boucles, l'une parcourant les pays et l'autre pour trouver les villes de chaque pays. En XSLT 2.0, grâce à l'instruction `for-each-group`, nous allons pouvoir tout faire en une itération.

```
<xsl:for-each-group group-by="@pays" select="villes/ville">
Pays : <xsl:value-of select="@pays"/>:
Villes : <xsl:value-of select="current-group()/@nom" separator=","/>
Population totale : <xsl:value-of select="sum(current-group()/@pop)"/>
</xsl:for-each-group>
```

On indique par l'attribut `group-by` sur quel critère on souhaite réaliser les groupes. On peut ensuite afficher ce critère par une expression XPath. La fonction `current-group` contient tous les éléments du groupe. Grâce à l'instruction `value-of` et à son attribut `separator`, il est possible d'afficher tous les noms de ville. Il en est de même pour la somme des populations.

On obtient donc comme résultat :

```
Pays : italy:
Villes : milan,venice
Population totale : 6

Pays : france:
Villes : paris,lyon
Population totale : 9

Pays : germany:
Villes : munich
Population totale : 4
```

Si nous avions souhaité afficher le détail de chaque ville, de chaque groupe, nous aurions pu écrire une deuxième boucle comme ci-après :

```
<xsl:for-each-group group-by="@pays" select="villes/ville">Pays :
➥<xsl:value-of select="@pays"/>:
<xsl:for-each select="current-group()">
Ville : <xsl:value-of select="@nom"/>
Population : <xsl:value-of select="@pop"/> Millions
</xsl:for-each>
Population totale : <xsl:value-of select="sum(current-group()/@pop)"/> Millions
</xsl:for-each-group>
```

Nous aurions alors obtenu ce résultat :

```
Pays : italy:
Ville : milan
Population : 5 Millions
```

```
Ville : venice
Population : 1 Millions
Population totale : 6 Millions

Pays : france:
Ville : paris
Population : 7 Millions
Ville : lyon
Population : 2 Millions
Population totale : 9 Millions

Pays : germany:
Ville : munich
Population : 4 Millions
Population totale : 4 Millions
```

La création de fonctions

L'un des points intéressants de XSLT 2.0 est la création de nouvelles fonctions. Cette fonctionnalité est également disponible dans XQuery. Une fonction peut avoir des arguments. Elle est appelée dans une expression XPath. Le type de la valeur retournée par la fonction et les types de paramètres sont spécifiés par l'attribut `as`. Les fonctions sont déclarées en tête de feuille de styles et, afin d'éviter des collisions entre les fonctions prédéfinies ou inclues, il est nécessaire de préfixer le nom des fonctions.

Voici un exemple :

```
<xsl:stylesheet exclude-result-prefixes="xs fn" version="2.0" xmlns:fn=
➥"http://www.w3.org/2005/xpath-functions" xmlns:fonc="http://www.masociete.com"
➥mlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsl=
➥"http://www.w3.org/1999/XSL/Transform">
<xsl:function as="xs:double" name="fonc:max">
 <xsl:param as="xs:double" name="v1"/>
 <xsl:param as="xs:double" name="v2"/>
 <xsl:value-of select="if ( $v1&gt;$v2 ) then $v1 else $v2"/>
</xsl:function>
<xsl:template match="/">
 <xsl:value-of select="fonc:max(10.0,11.4)"/>
</xsl:template>
</xsl:stylesheet>
```

Dans cet exemple, nous avons créé une fonction `max` dans l'espace de noms `http://www.masociete.com`. Cette fonction prend deux décimaux en argument et retourne la valeur maximale. À l'appel, il est aussi indispensable de spécifier l'espace de noms.

Utilisation des expressions régulières

Nous avons déjà vu qu'il était possible de réaliser certaines opérations avec XPath 2.0 en usant des expressions régulières via les fonctions `fn:matches`, `fn:replace` et `fn:tokenize`. XSLT 2.0 dispose également des instructions `analyze-string` et `matching-substring` pour

extraire certains blocs de texte à l'aide d'une expression régulière. L'instruction `non-matching-substring` sert lorsque la chaîne ne correspond pas à l'expression régulière. La fonction `regex-group`, avec pour argument un numéro de groupe supérieur ou égal à 1, sert à extraire le bloc de texte correspondant au groupe indiqué (pour rappel, un groupe est un ensemble de caractères entre parenthèses dans l'expression régulière).

Voici un exemple :

```
<xsl:analyze-string select="'10 juillet 07'" regex="(\d{{2}})\s+(\w+)\s(\d{{2}})">
 <xsl:matching-substring>
  <xsl:value-of select="regex-group(1)"></xsl:value-of>,
  <xsl:value-of select="regex-group(2)"></xsl:value-of><xsl:text> </xsl:text>
  <xsl:value-of select="concat(20,regex-group(3))"></xsl:value-of>
 </xsl:matching-substring>
 <xsl:non-matching-substring>
 Erreur dans la date ?
 </xsl:non-matching-substring>
</xsl:analyze-string>
```

Dans la première ligne, on définit l'expression régulière. À noter que nous sommes obligés de doubler les accolades, car une accolade simple est évaluée comme une expression XPath. Cette instruction régulière représente deux chiffres (`\d`), suivis d'au moins un blanc (`\s`), suivi d'un ensemble de caractères (`\w`), puis d'un blanc et de deux chiffres. Les parenthèses représentent trois groupes. L'instruction `matching-substring` est exécutée si la valeur présente dans l'attribut `select` correspond à cette expression régulière. Puis, chaque instruction `value-of` permet d'afficher une valeur issue de chaque groupe grâce à la fonction `regex-group`.

Nous obtenons comme résultat :

```
10,juillet 2007
```

Le format XSL-FO

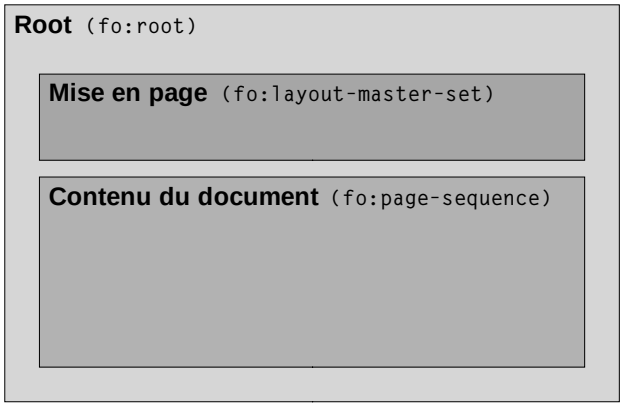
XSL-FO (*Extensible Stylesheet Language – Formatting Objects*) fait partie du standard XSL (dernière version 1.1, *http://www.w3.org/TR/xsl/*) au même titre que XSLT. Son rôle est d'offrir un format de présentation, comme HTML, mais servant à générer des formats plus complexes, comme les formats PDF, RTF et PostScript. XSLT n'aurait pas pu être utilisé en tant que tel pour ces formats, qui sont de structure trop complexe et possèdent souvent des données binaires (images, polices…). De plus, il aurait fallu créer autant de feuilles de styles que de formats. C'est pourquoi XSL-FO s 'emploie en document XML qui est le résultat d'une transformation XSLT. Ce document est ensuite converti par un programme dans un ou plusieurs formats.

Les éléments d'un document XSL-FO (ou FO) sont dans l'espace de noms `http://www.w3.org/1999/XSL/Format`. On a l'habitude d'utiliser le préfixe `fo` pour les qualifier.

Structure d'un document XSL-FO

Un document FO est composé d'instructions de mise en page (dimension…) et d'instructions d'affichage.

Figure 5-3
Structure d'un document FO



Dans la figure 5-3, nous avons donc un élément racine (`root`) contenant un en-tête de mise en page (`layout-master-set`) et une séquence de pages (`page-sequence`).

La séquence de pages est définie en effectuant une référence vers une mise en page. Voici un exemple :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
 <fo:simple-page-master master-name="A4">
 <!-- Mise en page -->
 </fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="A4">
 <!-- Contenu de page -->
</fo:page-sequence>
</fo:root>
```

Les attributs `master-name` et `master-reference` servent à établir ce lien.

La mise en page d'un document XSL-FO

La mise en page concerne les dimensions de chaque page, mais aussi le découpage en régions. Certaines régions seront statiques, c'est-à-dire seront toujours affichées de la même manière, comme le pied de page avec un numéro de page. D'autres régions seront dynamiques, c'est-à-dire que leur contenu sera défini page par page. Le changement de page sera alors effectué automatiquement lorsqu'une région sera pleine. Un saut de page explicite sera cependant possible.

Il y a deux instructions principales pour décrire un format de page :

- `simple-page-master` : décrit une page.
- `page-sequence-master` : décrit une séquence de pages, par exemple, en utilisant la parité du numéro de page (alignement à gauche ou à droite du numéro de page).

Le cas simple-page-master

Cette instruction contient des attributs pour définir les proportions de la page.

Voici un exemple :

```
<fo:simple-page-master
 master-name="A4"
 page-width="297mm"
 page-height="210mm"
 margin-top="1cm"
 margin-bottom="1cm"
 margin-left="1cm"
 margin-right="1cm">
</fo:simple-page-master>
```

Dans cet exemple, nous avons défini un format `A4` en paysage avec une marge de 1 cm. Les unités possibles sont le `cm`, `mm`, `in` (2.54`cm`), `pt` (1/72`in`), `pc` (12`pt`), `px` (pixels).

Une page peut être découpée en régions. Cela comprend une ou plusieurs parties centrales (`region-body`) et quatre régions potentielles autour des parties centrales (`region-before`, `region-after`, `region-start`, et `region-end`).

La figure 5-4 donne la localisation de ces différentes régions. La taille de ces régions est définie par l'attribut `extent`. Seules les régions centrales auront une taille en fonction de l'espace restant.

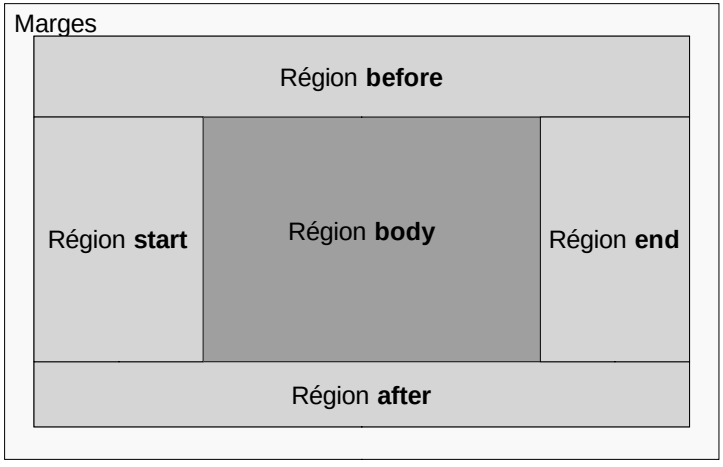


Figure 5-4
Répartition des régions

Voici un exemple comprenant quelques régions :

```
<fo:simple-page-master master-name="A4"
 page-width="297mm"
 page-height="210mm"
 margin-top="1cm"
 margin-bottom="1cm"
 margin-left="1cm"
 margin-right="1cm">
 <fo:region-body margin="3cm"/>
 <fo:region-start extent="2cm"/>
 <fo:region-before extent="2cm"/>
 <fo:region-after extent="2cm"/>
 <fo:region-end extent="2cm"/>
</fo:simple-page-master>
```

Remarque

La région centrale n'a pas de taille mais peut comporter une marge. Les régions situées autour ont une taille mais ne peuvent pas avoir de marge ; la propriété `padding` est alors utilisée.

Le cas page-sequence-master

Cette instruction sert à choisir un formatage de page en fonction d'un ordre ou d'une alternance de pages. Elle peut contenir des définitions de page simple, par l'instruction `simple-page-master`, ou bien permettre de forcer une présentation sur un ensemble de pages, par l'instruction `repeatable-page-master-reference`. L'instruction `repeatable-page-master-alternatives` effectue, quant à elle, un changement automatique à chaque page en fonction de certaines règles (parité de la page, page blanche…).

Voici un exemple :

```
<fo:layout-master-set>
<fo:simple-page-master master-name="debut" margin="3cm"  page-height="297mm"
➥page-width="210mm">
 <fo:region-body/>
</fo:simple-page-master>

<fo:simple-page-master master-name="suite" margin="1cm"  page-height="297mm"
➥page-width="210mm">
 <fo:region-body/>
</fo:simple-page-master>

<fo:simple-page-master master-name="fin" margin="2cm"  page-height="297mm"
➥page-width="210mm">
 <fo:region-body/>
</fo:simple-page-master>

<fo:page-sequence-master master-name="miseEnPage">
<fo:single-page-master-reference   master-reference="debut"/>
```

```
 <fo:repeatable-page-master-reference master-reference="suite"
 ➥maximum-repeats="10"/>
  <fo:single-page-master-reference   master-reference="fin"/>
</fo:page-sequence-master>

</fo:layout-master-set>
```

Dans cet exemple, nous avons défini trois formats de page, volontairement simplifiés, que nous avons appelés `debut`, `suite` et `fin`. Ces trois formats servent à définir une séquence de pages, via l'instruction `page-sequence-master`. Cette séquence commence pour la première page avec le format `debut`, puis se poursuit pendant les 10 pages qui suivent (grâce à l'attribut `maximum-repeats`) avec le format `suite`. Enfin, la dernière page (la douzième) est associée au format `fin`.

L'instruction `repeatable-page-master-alternatives` va contenir des instructions `conditional-page-master-reference`, indiquant le système d'alternance et effectuant une référence à un format de page par l'attribut `master-reference`.

Voici un exemple :

```
<fo:layout-master-set>
<fo:simple-page-master master-name="debut" margin="1cm"  page-height="297mm"
➥page-width="210mm">
 <fo:region-body/>
</fo:simple-page-master>
<fo:simple-page-master master-name="suite" margin="2cm"  page-height="297mm"
➥page-width="210mm">
 <fo:region-body/>
</fo:simple-page-master>
<fo:page-sequence-master master-name="miseEnPage">
 <fo:repeatable-page-master-alternatives>
  <fo:conditional-page-master-reference master-reference="debut"
  ➥odd-or-even="odd"/>
 <fo:conditional-page-master-reference master-reference="suite"
  ➥odd-or-even="even"/>
</fo:repeatable-page-master-alternatives>
</fo:page-sequence-master>
</fo:layout-master-set>
```

L'attribut `odd-or-even` prend la valeur `odd` pour un numéro de page impaire et `even` pour un numéro de page paire.

Intégration d'un contenu

Le contenu des pages sera défini avec l'instruction `page-sequence`. Cette instruction contiendra des données spécifiées à l'aide des instructions `flow` ou `static-content`. Cette dernière instruction servira uniquement aux données qui ne varient pas d'une page à l'autre, comme les en-têtes et pieds de page. Ces deux formes d'affichage de données pourront faire référence à une région par l'attribut `flow-name`.

Voici un exemple avec la présence d'un en-tête et d'un contenu :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
 <fo:simple-page-master master-name="page" margin="1cm"  page-height="297mm"
 ➥page-width="210mm">
  <fo:region-body margin="1cm"/>
  <fo:region-before extent="2cm"/>
 </fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="page">
 <fo:static-content flow-name="xsl-region-before">
  <fo:block>En-tête</fo:block>
 </fo:static-content>
 <fo:flow flow-name="xsl-region-body">
  <fo:block>Contenu</fo:block>
 </fo:flow>
</fo:page-sequence>
</fo:root>
```

L'attribut `flow-name` peut prendre, en fonction de la région souhaitée, les valeurs `xsl-region-body`, `xsl-region-start`, `xsl-region-end`, `xsl-region-before`, `xsl-region-after`, ou bien le nom d'une région (si l'attribut `region-name` est précisé lors de la définition de la région).

Analogie avec les paragraphes : les blocs

Le texte affiché dans un flux est réparti en blocs. Un bloc (élément `block`) peut représenter un ensemble de lignes (un peu comme un paragraphe) ou bien une partie de ligne (élément `inline`). On peut faire l'analogie avec les balises `div` et `span` en XHTML. Les attributs `space-before` et `space-after` servent à déterminer les écarts supplémentaires entre blocs.

Les propriétés graphiques des blocs sont identiques à celles du CSS 2.0 que nous avons déjà abordé.

Voici un exemple :

```
<fo:flow flow-name="xsl-region-body">
<fo:block
 color="#FFFFFF"
 background-color="green"
 border-color="red"
 border-style="solid"
 border-width="2">Test</fo:block>
</fo:flow>
```

Ce code fera apparaître le mot `Test` avec une bordure rouge et un fond vert.

Voici un autre exemple :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
 <fo:simple-page-master master-name="A4" margin="1cm">
  <fo:region-body margin="1cm"/>
 </fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="A4">
 <fo:flow flow-name="xsl-region-body">
 <fo:block><fo:inline color="red">T</fo:inline>est </fo:block>
 </fo:flow>
</fo:page-sequence>
</fo:root>
```

Cette fois-ci, la lettre `T` sera affichée en rouge dans le mot `Test`.

Il est possible de réaliser un saut de page en début ou fin de bloc, respectivement avec les attributs `page-break-before` et `page-break-after`. Ces attributs peuvent prendre les valeurs suivantes :

- `auto` : par défaut ;
- `always` : insère toujours un saut de page ;
- `avoid` : évite un saut de page ;
- `left` : saute une ou deux pages pour que la prochaine page soit à gauche ;
- `right` : saute une ou deux pages pour que la prochaine page soit à droite.

La représentation de listes

Les listes (à puces) sont représentées par l'instruction `list-block`. Chaque puce est associée à un élément `list-item`. Dans chaque puce, on décrit la zone de gauche (pastille, énumération) grâce à l'instruction `list-item-label`, la zone principale étant définie dans l'instruction `list-item-body`.

Voici un exemple :

```
<fo:list-block>
<fo:list-item>
 <fo:list-item-label><fo:block>*</fo:block></fo:list-item-label>
 <fo:list-item-body start-indent="body-start()"><fo:block>ITEM 1</fo:block>
 ➥</fo:list-item-body></fo:list-item>
...
</fo:list-block>
```

L'attribut `start-indent` sert à indenter le texte de chaque puce.

L'intégration de tableaux

Un tableau est compris dans l'instruction `table-and-caption`. Le titre du tableau est défini par l'instruction `table-caption`. Le corps principal est présent dans l'instruction `table`.

Les dimensions de chaque colonne sont décrites par l'instruction `table-column` ; les entêtes, fin de tableau et parties principales sont respectivement définies à l'aide des instructions `table-header`, `table-footer` et `table-body`. Ces trois dernières parties sont découpées en lignes par l'instruction `table-row`. Chaque ligne est elle-même découpée en colonnes par l'instruction `table-cell`.

Voici un exemple qui vous montre l'imbrication des ces instructions :

```
<fo:table-and-caption>
 <fo:table-caption>
  <fo:block>Titre du tableau</fo:block>
 </fo:table-caption>
 <fo:table table-layout="fixed" width="200px">
  <fo:table-column column-number="1" column-width="100px"/>
  <fo:table-column column-number="2" column-width="100px"/>
  <fo:table-header>
   <fo:table-row>
    <fo:table-cell>
     <fo:block>Colonne 1</fo:block>
    </fo:table-cell>
    <fo:table-cell>
     <fo:block>Colonne 2</fo:block>
    </fo:table-cell>
   </fo:table-row>
  </fo:table-header>
 <fo:table-body>
 <fo:table-row>
   <fo:table-cell>
    <fo:block>A</fo:block>
   </fo:table-cell>
   <fo:table-cell>
    <fo:block>B</fo:block>
   </fo:table-cell>
  </fo:table-row>
 </fo:table-body>
 </fo:table>
</fo:table-and-caption>
```

L'attribut de table `table-layout` indique comment l'algorithme doit procéder pour effectuer le rendu de la table. Lorsque la valeur `fixed` est employée, le rendu de la table n'est pas réalisé en fonction du contenu des cellules, mais uniquement en fonction des dimensions des colonnes (propriété `width`) et des autres espaces précisés (marge interne, bordure…). Si la valeur `auto` est utilisée, l'algorithme se base sur l'occupation des données dans chaque cellule afin d'ajuster au mieux les dimensions des colonnes.

Remarque à propos de FOP

Pour utiliser une table avec FOP (v0.93), il ne faut pas utiliser l'élément `table-and-caption` mais seulement l'élément `table`.

L'occupation par une cellule de plusieurs colonnes ou de plusieurs lignes peut être réalisée respectivement par les attributs `number-columns-spanned` et `number-rows-spanned`.

L'insertion d'images

L'incorporation d'une image peut être menée de différentes manières. La première solution consiste à utiliser l'élément `external-graphic` avec l'attribut `src` pour indiquer la localisation de l'image à insérer. La deuxième solution s'appuie sur la propriété CSS `background-image`.

Voici un exemple :

```
<fo:page-sequence master-reference="page">
 <fo:flow flow-name="xsl-region-body">
  <fo:block>
   <fo:external-graphic src="c:/plan.gif"/>
  </fo:block>
  <fo:block background-image="c:/plan.gif">
 Hello world
  </fo:block>
 </fo:flow>
</fo:page-sequence>
```

Remarque à propos de FOP

La version 0.93 ne gère pas les chemins relatifs (d'où la présence de `c:/`).

Les dimensions de l'image peuvent être modifiées avec les attributs `content-width` (largeur) et `content-height` (hauteur).

L'insertion de figures SVG

Nous avons intégré, dans les cas précédents, des images bitmap, ce qui n'est pas toujours souhaitable ou réalisable dans un contexte multidocument. On peut, par exemple, imaginer que l'on ait aussi besoin de produire un graphe en fonction de certains résultats. Dans ce cas, il existe la possibilité de passer par le format SVG (*Scalable Vector Graphics*). Ce format dispose de nombreuses primitives pour réaliser des représentations 2D. Nous aborderons le détail de ce langage dans le prochain chapitre ; voici un exemple mêlant XSL-FO et SVG :

```
<fo:block>
 <svg:svg
width="200pt"
height="200pt"
xmlns:svg="http://www.w3.org/2000/svg">
  <svg:circle cx="100pt" cy="100pt" r="90pt" style="fill:blue;"/>
 </svg:svg>
</fo:block>
```

Cohabitation entre XSLT et XSL-FO : la chaîne complète

Pour transformer un document XML en un document PDF, par exemple, nous devons générer, par une transformation XSLT, un document XSL-FO. Viendra alors une dernière étape pour générer le document final.

Voici un exemple qui transforme notre document XML composé de personnes (avec un attribut `nom`) en document XSL-FO :

```
<xsl:stylesheet
version="1.0"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output indent="yes" method="xml"/>
<xsl:template match="/">
 <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
   <fo:simple-page-master master-name="page">
    <fo:region-body margin="1cm"/>
   </fo:simple-page-master>
  </fo:layout-master-set>
  <fo:page-sequence master-reference="page">
   <fo:flow flow-name="xsl-region-body">
    <xsl:apply-templates select="//personne"/>
   </fo:flow>
  </fo:page-sequence>
 </fo:root>
</xsl:template>

<xsl:template match="personne">
 <fo:block><xsl:value-of select="@nom"/></fo:block>
</xsl:template>
</xsl:stylesheet>
```

Ce qui peut produire, par exemple, ce document (avec deux personnes) :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
<fo:simple-page-master master-name="page">
 <fo:region-body margin="1cm"/>
</fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="page">
 <fo:flow flow-name="xsl-region-body">
  <fo:block>dupont</fo:block>
  <fo:block>dupond</fo:block>
 </fo:flow>
</fo:page-sequence>
</fo:root>
```

Il reste à effectuer la dernière étape pour convertir ce document au format choisi, ce qui peut être réalisé, par exemple, en utilisant FOP (*http://xmlgraphics.apache.org/fop/*).

Les liens entre documents

Les liens entre documents peuvent être externes ou internes, respectivement selon les attributs `external-destination` ou `internal-destination` de l'élément `basic-link`. Pour effectuer un lien interne, il suffit de spécifier la valeur d'un attribut `id` positionnée, par exemple, sur un `block`.

Voici un exemple pour chaque type de lien :

```
<fo:flow flow-name="xsl-region-body">
 <fo:block>
  <fo:basic-link
   external-destination="http://www.google.com"
   color="blue"
   border-after-style="solid"
   border-after-color="blue">http://www.google.com</fo:basic-link>
  <fo:basic-link
    internal-destination="A1">Vers Ancre</fo:basic-link>
 </fo:block>
 <fo:block page-break-before="always" id="A1">Ancre</fo:block>
</fo:flow>
```

Dans le premier lien, nous allons directement sur la page du site de Google. Nous avons ajouté des propriétés graphiques pour que le lien apparaisse souligné en bleu. Dans le second lien, nous changeons de page et nous dirigeons automatiquement vers la partie contenant l'identifiant `A1`.

Exercice 9

À partir du document de l'exercice 4, réalisez une transformation XSLT pour produire un document XSL-FO affichant la table des matières. Vous effectuerez ensuite la génération d'un document PDF.

Le format vectoriel SVG

SVG (*Scalable Vector Graphics*) sert à représenter un dessin en 2D via un ensemble de primitives formalisées en XML. La dernière version est la version 1.1 (*http://www.w3.org/TR/SVG11/*) mais une version 1.2 est en cours de préparation au moment de la rédaction de cet ouvrage.

Les éléments SVG sont situés dans l'espace de noms `http://www.w3.org/2000/svg`. On sauvegarde généralement ce type de document avec l'extension `.svg`. Les éléments de rendu sont soit des figures géométriques (rectangle, cercle…), soit du texte. On agit, dans la représentation de ces éléments, soit au niveau de leur bordure (`stroke`), soit au niveau

du contenu (`fill`). Différents modes de rendu peuvent influer sur la couleur, un dégradé (gradient) ou un motif répété (pattern, comme les pointillés).

La plupart des propriétés CSS sont disponibles, soit via l'attribut `style`, soit sous la forme d'un attribut par propriété.

Les éléments SVG peuvent posséder un identifiant unique via l'attribut `id`.

Les différentes figures géométriques

Les figures géométriques sont positionnées dans un repère dont l'origine est en haut à gauche. Les unités disponibles sont `em` (hauteur de la fonte), `ex` (hauteur liée à la propriété CSS `font-height`), `px` (pixel), `pt` (1.25px), `pc` (15px), `cm` (35.43307px), `mm`, `in` (90px) et les pourcentages.

Voici quelques propriétés d'usage courant :

- `fill` : remplissage de la figure ;
- `fill-opacity` : degré de transparence pour le remplissage (entre 0 et 1) ;
- `opacity` : degré de transparence (entre 0 et 1) ;
- `stroke` : bordure de la figure ;
- `stroke-width` : épaisseur de la bordure ;
- `stroke-opacity` : degré de transparence pour la bordure (entre 0 et 1).

La représentation du rectangle

Le rectangle est construit par l'élément `rect`. Il est défini par la position `x` et `y` alors que ses proportions sont définies par les attributs `width` et `height`. Les attributs `rx` et `ry` servent à définir une bordure arrondie. Voici un exemple :

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <rect x="10%" y="10%" width="80%" height="80%"
 style="fill:blue;stroke:pink;stroke-width:5;
 fill-opacity:1;stroke-opacity:0.9"/>
</svg>
```

Ici, nous avons représenté un rectangle qui occupe 80 % de l'espace disponible. L'opacité est maximale (1) et une bordure de 5 pixels, de couleur rose foncé, l'entoure.

La représentation du cercle

Le cercle est construit par l'élément `circle`. Les attributs `cx`, `cy` et `r` désignent respectivement la position du centre du cercle et son rayon.

Voici un exemple créant un cercle rouge avec une bordure noire :

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <circle cx="50" cy="50" r="40" stroke="black" stroke-width="2" fill="#FF0000"/>
</svg>
```

Similaire au cercle, une ellipse est définie, en plus des attributs `cx` et `cy`, par les attributs `rx` et `ry`, qui représentent ses proportions (horizontale et verticale). Bien sûr, si ces valeurs sont identiques nous obtenons un cercle, comme dans cet exemple :

```
<ellipse cx="50" cy="50" rx="40" ry="40" stroke="black"
stroke-width="2" fill="#FF0000"/>
```

La représentation de la ligne

La ligne est définie par l'élément `line`. Les attributs `x1`, `y1` et `x2`, `y2` définissent les coordonnées de ses deux extrémités.

Voici un exemple avec une ligne qui occupe une diagonale :

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <line x1="10%" y1="20%" x2="80%" y2="60%" style=
 ➥"stroke:black;stroke-width:1px"></line>
</svg>
```

La représentation du polygone

Le polygone est associé à l'élément `polygon` et est constitué d'au moins trois points (reliés par une ligne). L'attribut `points` contient l'ensemble de ces coordonnées, séparées par un espace.

Voici un exemple affichant un trapèze :

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <polygon points="10,10 100,10 150,100 1,100"></polygon>
</svg>
```

L'élément `polygon` créé une figure fermée (le dernier point est connecté au premier point). L'élément `polyline` n'a pas cette caractéristique et peut, par exemple, servir à la réalisation de graphes.

La représentation d'un chemin

La notion de path ou chemin est particulière en SVG : elle sert à réaliser des figures complexes grâce à un ensemble de coordonnées et de commandes qui précisent les relations entre les points (lignes, courbes...). Voici les commandes disponibles :

- `M` : déplacement (*Move*) ;
- `L` : création d'une ligne (*Line*) ;
- `H` : création d'une ligne horizontale (*Horizontal line*) ;
- `V` : création d'une ligne verticale (*Vertical line*) ;
- `C` : création d'une courbe (*Curve*) ;
- `S` : création d'une courbe lissée (*Smooth*) ;

- `Q` : création d'une courbe de bézier ;
- `T` : création d'une courbe de bézier lissée ;
- `A` : création d'un arc de cercle ;
- `Z` : terminaison du chemin ou path.

Chaque commande (à part la terminaison) est suivie de 1 ou de plusieurs points. Attention, les coordonnées de chaque point n'ont pas de virgule.

Voici un exemple :

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <path d="M10 10 L100 100 C200 200 250 220 180 220 Z" />
</svg>
```

Cette figure est la juxtaposition d'une ligne et d'une courbe.

La réutilisation des définitions

Les éléments graphiques peuvent être définis, un peu à la manière d'une déclaration de fonction, et utilisés dans divers contextes, avec l'avantage de posséder une définition unique. L'élément `defs` sert à stocker la définition de ces éléments alors que l'élément `use` donne la possibilité de les exploiter (sur une position nouvelle, par exemple). Le lien entre l'élément défini et son utilisation est réalisé avec XLink (*http://www.w3.org/TR/xlink/*).

Voici un exemple :

```
<svg
width="100%"
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
>
<defs>
<circle
id="cercle1" r="50" style="fill:red;stroke:black"></circle>
<circle
id="cercle2" r="20" style="fill:black;stroke:red"></circle>
</defs>

<use xlink:href="#cercle1" x="60" y="60"></use>
<use xlink:href="#cercle2" x="30" y="30"></use>

</svg>
```

Ici, nous avons employé les définitions `cercle1` et `cercle2`.

Le rendu du texte

Le texte est rendu grâce à l'élément `text`. Il n'y a pas de retour à la ligne automatique si l'espace restant n'est pas suffisant pour contenir le texte. Le texte est positionné grâce aux attributs `x` et `y`.

Voici un exemple :

```
<svg
width="100%"
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg">
 <text x="30" y="70" style="font-size:50pt;font-family:times;stroke:red;
 ➥stroke-width:2px;fill:green">Hello world</text>
</svg>
```

Dans cet exemple, le texte est grossi grâce au style CSS `font-size`, nous avons retenu une police avec empattement `times`, les caractères sont remplis en vert et ont une bordure rouge.

L'alignement du texte peut être géré avec l'attribut `text-anchor`. La valeur n'aura de sens que par rapport à l'orientation de l'affichage (du haut vers le bas, de la gauche vers la droite…). On emploie donc les valeurs `none`, `start` (début), `middle` (centré) ou `end` (fin) à cet effet.

On peut remplacer les attributs `x` et `y` par les attributs `dx` et `dy`, qui présentent l'avantage de gérer la position relativement au dernier texte. Pour note, cette fonctionnalité n'est pas disponible sous Firefox 2.0.

Rappel de quelques propriétés CSS pour le texte :

- `font-family` : police de caractères employée.
- `font-style` : `normal`, `italic`, ou `oblique`.
- `font-variant` : deux possibilités pour les minuscules `normal` ou `small-caps`.
- `font-weight` : indique le niveau de gras selon `normal`, `bold`, `bolder`, `lighter`, `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, `900`.
- `font-stretch` : représente l'écart moyen entre les caractères selon `normal`, `wider`, `narrower`, `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `semi-expanded`, `expanded`, `extra-expanded`, `ultra-expanded`.
- `font-size` : taille des caractères (en valeur relative ou absolue).

Le texte peut être aligné sur un path grâce à l'élément `textPath`. Le lien entre le texte aligné et le path est réalisé avec XLink.

Voici un exemple :

```
<svg
width="100%"
```

```
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
>
 <defs>
  <path id="courbe"
  d="M 100 200
     C 200 100 300   0 400 100
     C 500 200 600 300 700 200
     C 800 100 900 100 900 100" />
 </defs>

 <text font-family="Verdana" font-size="42.5" fill="blue" >
  <textPath xlink:href="#courbe">
   Démonstration d'alignement sur un path
 </textPath>
 </text>

</svg>
```

Dans cet exemple, le texte est associé à la définition d'un path. Il aurait été aussi possible d'afficher la courbe liée avec l'élément `use`.

L'élément `tspan`, un peu à la manière de la balise `span` en XHTML/HTML, sert à modifier graphiquement un contenu de ligne.

Voici un exemple :

```
 <svg
width="100%"
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg"
>

<text x="20" y="20">
 <tspan>Hello</tspan>
 <tspan dy="10" style="fill:red">World</tspan>
</text>

</svg>
```

Les mots `Hello` et `World` sont légèrement décalés de par l'attribut `dy` qui applique un déplacement relatif vertical ; le dernier mot est en rouge grâce à la propriété `fill:red`.

Quelques effets graphiques

Nous présentons ici quelques effets qui peuvent s'ajouter aux figures géométriques.

Le dégradé linéaire

Un dégradé remplit la figure en passant d'une couleur à une autre. Ce passage est effectué, dans un dégradé linéaire, par une superposition de lignes. Cette forme de dégradé est réalisée par l'élément `linearGradient`. Ce dernier contient au moins une couleur de départ et une couleur de fin avec deux éléments `stop`. Chaque élément `stop` indique l'endroit où la couleur de départ ou de fin s'arrête (attribut `offset`) et sa valeur.

Voici un exemple avec un dégradé linéaire du blanc au noir :

```
<defs>
  <linearGradient id="deg1">
    <stop offset="5%" stop-color="white" />
    <stop offset="85%" stop-color="black" />
  </linearGradient>
</defs>

<rect width="100" height="100" style="fill:url('#deg1')"></rect>
```

Dans ce cas, nous avons une bande blanche qui occupe 5 % de la figure ; puis une somme de couleurs transitoires est effectuée jusqu'à 85 % de la figure, qui passe alors à la couleur noire.

L'orientation des lignes du dégradé peut être réalisée par les attributs `x1`, `y1` et `x2`, `y2` représentant deux points en pourcentage.

Voici un exemple avec une ligne de dégradé diagonale :

```
<defs>
 <linearGradient id="deg1" x1="1%" y1="1%" x2="90%" y2="90%">
  <stop offset="0%" stop-color="white" />
  <stop offset="100%" stop-color="black" />
 </linearGradient>
</defs>
```

Le dégradé radial

Dans un dégradé radial, les lignes sont remplacées par des cercles de rayon croissant. Il est possible de modifier le centre de chaque cercle. L'élément `radialGradient` se gère un peu comme une figure `circle` : le centre du cercle avec les attributs `cx` et `cy`, et la taille du dégradé par l'attribut `r`.

Voici un exemple :

```
<svg
width="100%"
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg"
>
```

```
<defs>
 <radialGradient  id="deg1" cx="50%" cy="50%" r="60%" >
  <stop offset="0%" stop-color="black" />
  <stop offset="90%" stop-color="white" />
 </radialGradient>
 </defs>
 <rect width="100" height="100" style="fill:url('#deg1')"></rect>
</svg>
```

L'interactivité avec les figures SVG par scripting

Un dessin SVG peut, comme une page XHTML/HTML, être interactif. Cela fonctionne un peu sur le même principe, un élément SVG pouvant être associé à des événements utilisateur (par exemple `onclick`) et provoquer l'exécution d'un morceau de code.

Voici un exemple avec du JavaScript :

```
<svg width="100%"
height="100%"
version="1.1"
xmlns="http://www.w3.org/2000/svg" version="1.1">
  <script type="text/ecmascript"> <![CDATA[
    function un_click(evt) {
      var cercle = evt.target;
      var rayon = cercle.getAttribute( "r" );
      if (rayon == 100)
        cercle.setAttribute( "r", rayon *2 );
      else
        cercle.setAttribute("r", rayon *0.5);
    }
  ]]> </script>
  <circle onclick="un_click(evt)" cx="300" cy="225" r="100"
          fill="red"/>
</svg>
```

Si nous cliquons sur le cercle, la fonction JavaScript `un_click` est alors appelée. Une référence à l'objet déclencheur est alors récupérée par l'instruction `evt.target`. Il reste à effectuer une modification du rayon `r` via l'API DOM (voir le chapitre consacré à la programmation).

Correction des exercices

L'ensemble des exercices a été réalisé sur le logiciel EditiX (*http://www.editix.com/*). Une version d'évaluation de 30 jours est librement téléchargeable (*http://www.editix.com/download.html*).

Exercice 1

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<h1>Table des matières</h1>
<ul>
 <li><a href="#tp1">TP1</a></li>
 <li><a href="#tp2">TP2</a></li>
</ul>
<hr/>
<a name="tp1"/>
 <h1>TP1</h1>
 <p>Détail  TP1 : </p>
 <a href="tp1.html">Lien</a>
 <a name="tp2"/>
  <h1>TP2</h1>
  <p>Détail  TP2 : </p>
  <a href="tp2.html">Lien</a>
 </body>
</html>
```

Exercice 2

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
 <table border="1">
  <tr>
   <td bgcolor="red">1,1</td>
   <td>1,2</td>
   <td>1,3</td>
  </tr>
  <tr>
   <td>2,1</td>
   <td bgcolor="red">2,2</td>
   <td>2,3</td>
  </tr>
  <tr>
   <td>3,1</td>
   <td>3,2</td>
   <td bgcolor="red">3,3</td>
  </tr>
  <tr>
   <td colspan="3">Matrice 3x3</td>
  </tr>
 </table>
</body>
</html>
```

Exercice 3

Voici le fichier CSS externe `style.css` :

```
body {
 font-family:Arial, Helvetica, sans-serif;
}
h1 {
 color:red;
 font-style:italic;
}
/* Image pour les puces */
ul {
 list-style-image:url( 'images/balle.jpg' );
}
a {
 color:green;
 font-weight:bold;
}
a:hover {
 color:red;
}
```

Nous avons ajouté les instructions suivantes avant la balise `body` de notre page XHTML/HTML de test :

```
<head>
    <link rel="stylesheet" href="tp5.css">
</head>
```

Exercice 4

Il existe plusieurs possibilités, à chaque fois plus ou moins optimales, sachant que l'on peut jouer sur le nombre de parcours nécessaires dans l'arbre.

– Trouver la liste des chapitres de la première section :

```
/child::livre/child::sections/child::section[1]/child::chapitre
```

– Trouver la liste des attributs du premier auteur :

```
/child::livre/child::auteurs/child::auteur[1]/attribute::*
```

– Trouver la valeur de l'attribut `nom` du deuxième auteur :

```
string( /child::livre/child::auteurs/child::auteur[2]/attribute::nom )
```

– Trouver la liste des chapitres contenant deux paragraphes :

```
/descendant::chapitre[count(child::paragraphe)=2]
```

– Trouver la liste des chapitres dont un paragraphe possède le mot `Premier` :

```
/descendant::chapitre[contains(child::paragraphe,'Premier')]
```

– Trouver la liste des sections ayant un chapitre :

```
/descendant::section[count(child::chapitre)=1]
```

– Trouver la liste des éléments ayant un seul attribut :

```
/descendant::*[count(attribute::*)=1]
```

– Trouver la liste des éléments ayant un ancêtre `sections` :

1. `/child::livre/child::sections/descendant::*`
2. `/descendant::*[ancestor::sections]`

– Trouver la liste des attributs `titre` :

```
/descendant::*/attribute::titre
```

– Trouver la liste des éléments ayant deux fils et pas d'attribut :

```
/descendant-or-self::*[count(child::*)=2 and count(attribute::*)=0]
```

– Trouver la liste des `sections` sans paragraphe :

```
/descendant::section[count(descandant::paragraphe)=0]
```

– Trouver la liste des éléments dont le texte contient le mot `paragraphe` :

```
//descendant::text()[contains(self::text(), 'paragraphe')]/parent::*
```

Exercice 5

– Trouver la liste de nœuds `auteur` :

```
//auteur
```

– Trouver la liste de tous les nœuds `section` :

```
//section
```

– Trouver la liste des chapitres de la première section :

```
/livre/sections/section[1]/chapitre
```

– Trouver la liste des attributs du premier auteur :

```
/livre/auteurs/auteur[1]/@*
```

– Trouver la valeur de l'attribut `nom` du deuxième auteur :

```
string( /livre/auteurs/auteur[2]/@nom )
```

– Trouver la liste des sections avec deux chapitres :

```
//section[count(chapitre)=2]
```

– Trouver la liste des paragraphes dont le parent a pour titre `Chapitre1` :

```
//paragraphe[../@titre='Chapitre1']
```

Exercice 6

– Afficher la liste des chapitres avec plus de deux paragraphes à l'aide d'une expression FLWOR :

```
for $ch in //chapitre where count($ch/*) > 2
return concat( $ch/@titre, ' ' )
```

– Afficher une séquence de texte avec la liste des sections et les chapitres liés :

```
for $sec in //section
return
(
'&#10;',
string( $sec/@titre ),
'&#10;',
for $ch in $sec/chapitre
return string( $ch/@titre )
)
```

Nous avons ajouté des entités `
` pour forcer un retour à la ligne sur chaque section. La séquence n'est composée que de chaînes ; nous avons donc enrobé chaque valeur par la fonction `string`.

Exercice 7

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

<xsl:template match="/livre">
 <html>
  <body>
   <xsl:apply-templates select="sections" mode="TM"/>
   <xsl:apply-templates select="sections" mode="FULL"/>
  </body>
 </html>
</xsl:template>

<xsl:template match="sections" mode="TM">
<xsl:for-each select="section">
  <a href="#{@titre}">
   <xsl:number/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@titre"/>
  </a>
  <br />
  <xsl:for-each select="chapitre">
   <a href="#{@titre}">
    <xsl:number count="section|chapitre" level="multiple"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@titre"/>
   </a>
   <br />
```

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

<xsl:template match="sections" mode="FULL">
 <xsl:for-each select="section">
  <a name="{@titre}">
   <h1>
  <xsl:number/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@titre"/>
   </h1>
  </a>
  <xsl:for-each select="chapitre">
   <a name="{@titre}">
    <h2>
     <xsl:number count="section|chapitre" level="multiple"/>
     <xsl:text> </xsl:text>
     <xsl:value-of select="@titre"/>
    </h2>
   </a>
   <xsl:apply-templates select="paragraphe"/>
  </xsl:for-each>
 </xsl:for-each>
</xsl:template>

<xsl:template match="paragraphe">
 <p>
  <xsl:value-of select="."/>
 </p>
</xsl:template>
</xsl:stylesheet>
```

Exercice 8

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" version="1.0" encoding="iso-8859-1" indent="yes"/>

<xsl:template match="/">
 <livre>
  <titre>
   <xsl:value-of select="livre/@titre"/>
  </titre>
  <auteurs>
   <xsl:for-each select="livre/auteurs/*">
    <auteur>
     <xsl:attribute name="nomPrenom">
      <xsl:value-of select="@nom"/>-<xsl:value-of select="@prenom"/>
     </xsl:attribute>
    </auteur>
   </xsl:for-each>
  </auteurs>
  <xsl:for-each select="//section">
```

```
   <section>
    <titre><xsl:value-of select="@titre"/></titre>
    <xsl:for-each select="chapitre">
     <chapitre>
      <titre><xsl:value-of select="@titre"/></titre>
      <xsl:for-each select="*">
       <para>
        <xsl:value-of select="."/>
       </para>
      </xsl:for-each>
     </chapitre>
    </xsl:for-each>
   </section>
  </xsl:for-each>
</livre>
</xsl:template>
</xsl:stylesheet>
```

Exercice 9

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
➥xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:output method="xml" indent="yes" encoding="iso-8859-1"/>

<xsl:template match="/">
 <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
   <fo:simple-page-master master-name="A4TM" margin="1cm">
    <fo:region-body margin-right="15cm" margin-left="1cm"/>
   </fo:simple-page-master>
   <fo:simple-page-master master-name="A4Body" margin="1cm">
    <fo:region-body margin-left="1cm"/>
    <fo:region-after extent="3cm"/>
   </fo:simple-page-master>
  </fo:layout-master-set>
  <xsl:apply-templates select="//section"/>
 </fo:root>
</xsl:template>
<xsl:template match="section">
<fo:page-sequence master-reference="A4Body">
 <fo:static-content flow-name="xsl-region-after">
  <fo:block text-align="end">
  <fo:page-number/>
  </fo:block>
 </fo:static-content>
 <fo:flow flow-name="xsl-region-body">
  <fo:block color="red" font-size="20pt">
   <xsl:value-of select="@titre"/>
   <xsl:for-each select="chapitre">
```

```
    <fo:block color="red" font-size="20pt">
     <xsl:value-of select="@titre"/>
     <xsl:for-each select="*">
      <fo:block color="black" font-size="12pt">
       <xsl:value-of select="."/>
      </fo:block>
     </xsl:for-each>
    </fo:block>
   </xsl:for-each>
  </fo:block>
 </fo:flow>
</fo:page-sequence>
</xsl:template>
</xsl:stylesheet>
```

OUTIL Scenari, un exemple de chaîne éditoriale XML libre

Scenari peut être vu comme un outil de génération de chaîne XML (qui génère donc du code RelaxNG, avec un outil d'édition en XUL, des moteurs de transformation XSLT, etc.) et un environnement d'édition/publication intégré qui « joue » ce code généré. Ce progiciel libre destiné à la création de chaînes éditoriales XML se compose d'un environnement de développement déclaratif, ScenariBuilder, et d'un environnement d'exécution intégré, ScenariChain.

Une chaîne XML peut-être vue – en simplifiant un peu – comme un assemblage :

– d'un schéma posant le langage documentaire (DTD, XML Schema, RelaxNG, etc.) ;

– d'un éditeur assurant la validité des contenus XML produits lors du schéma documentaire (Editix, XMLSpy, etc.) ;

– d'un ou plusieurs moteurs de publication permettant de transformer les contenus XML en contenus lisibles (programme XSLT vers HTML ou OpenDocumentFormat, par exemple).

ScenariBuilder est un environnement de haut niveau permettant de réaliser ses modèles, éditeurs et moteurs de publication, sans développement informatique. Le système prend en charge la génération des composants de la chaîne (schémas RelaxNg, interfaces d'édition, programmes XSLT, etc.) à partir d'un paramétrage déclaratif.

ScenariChain est une application intégrée prenant en entrée un pack ScenariBuilder et offrant une interface d'édition et des moteurs de publication (HTML, ODF, etc.) adaptés au modèle documentaire. La réalisation d'une chaîne XML avec Scenari est beaucoup plus rapide. C'est l'apport de tout progiciel qui factorise les développements. L'ambition du projet et de faciliter – par la réduction drastique des coûts de développements – la démocratisation de l'usage des outils XML pour la production et la publication de contenus structurés.

Références :

http://scenari-platform.org/

Stéphane Crozat, *Scenari, la chaîne éditoriale libre*, Eyrolles 2007 (collection Accès Libre).

6

Les échanges XML

Ce chapitre concerne les échanges de données XML entre applications. XML est un format universel qui permet de structurer des données texte, ce qui s'avère être une qualité lors de la communication entre applications hétérogènes (plate-forme, système d'exploitation et langage de programmation différents).

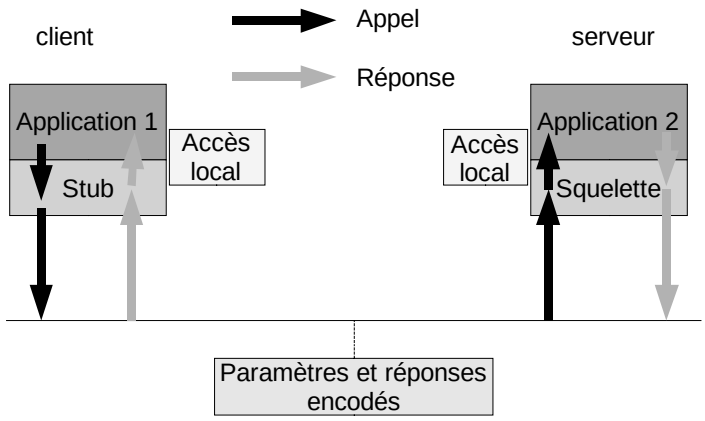
Son rôle dans l'entreprise

Comme nous l'avons vu, XML est une manière confortable de conserver des briques de données. La circulation des données est courante dans une entreprise où les activités sont rarement localisées à un même endroit. Chaque service, chaque filiale, chaque partenaire peut donner naissance à des échanges de données. Tout le problème est de disposer des moyens et d'une infrastructure capables d'effectuer ces échanges. En effet, si un choix technologique a été effectué à un endroit, il n'est pas certain que les mêmes choix aient été retenus ailleurs. XML joue alors un rôle important en garantissant un contexte universel compréhensible sur la plupart des plates-formes et par différents langages de programmation.

Le standard CORBA (*Common Object Request Broker Architecture*) a tenté de jouer ce rôle en mettant en place des bus de données reliant les différents composants d'une plate-forme. La faible percée de cette technologie a été due à sa trop grande complexité et aux contraintes imposées au développeur (limitation des types…). Dans ce standard, la communication est fondée sur l'idée qu'un appel de méthode dans un langage de programmation peut être réalisé par un programme distant, en effectuant une opération

de traduction des paramètres et de la valeur retour de la méthode. On le voit bien, cette traduction des paramètres et de la valeur de retour ainsi que la circulation de ces valeurs entre des programmes distants nécessitent un cadre commun de stockage lors du transport. XML a donc été intégré petit à petit dans ce système, tout d'abord avec les XML-RPC (*Remote Procedure Call*) et maintenant avec les services web et même Ajax (*Asynchronous JavaScript and XML*) dans un contexte orienté Internet/intranet.

Figure 6-1
Répartition des rôles



La figure 6-1 représente de quelle façon deux applications peuvent travailler ensemble. La première application cliente (application 1) dispose de *stubs*. Un stub joue le rôle d'intermédiaire entre l'application locale et l'application distante ; il intègre des méthodes mais ne réalise pas de traitement, à l'exception d'un transfert distant des paramètres à un squelette (*skeleton*), qui se charge d'appeler cette même méthode dans l'application serveur (application 2) et de retourner une valeur éventuelle (voire un message d'erreur). Bien souvent, l'usage du stub et du squelette revient à définir une interface (IDL : *Interface Description Language*) décrivant l'ensemble des signatures des méthodes que l'on peut utiliser. Des générateurs de code se chargent ensuite de la création du stub et squelette. Le terme squelette vient du fait que le code produit, bien qu'utilisable mais sans effet (méthodes vides), devra nécessairement être complété pour réaliser les traitements côté serveur.

Les échanges XML-RPC

RPC (*Remote Procedure Call*) est une technologie qui sert à invoquer une routine distante sans que le développeur ait besoin de détailler les échanges nécessaires. En programmation objet, elle peut être semblable à la technologie Java RMI (*Remote Method Invocation*). XML-RPC joue le même rôle en s'appuyant sur l'universalité de XML.

Les principes de XML-RPC

La technologie XML-RPC repose sur XML pour le stockage des données et sur HTTP pour le transport. Dans XML-RPC les types de données ont été traduits en XML. Voici quelques exemples :

```
Tableau :
<array>
  <data>
    <value><i4>1404</i4></value>
    <value><string>Une valeur</string></value>
    <value><i4>1</i4></value>
  </data>
</array>
Base64 (pour le binaire ) :
<base64>eW91IGNhbidOIHJlYWQgdGhpcyE=</base64>
Booléen :
<boolean>1</boolean>
Date/Heure :
<dateTime.iso8601>19980717T14:08:55</dateTime.iso8601>
Décimal :
<double>-12.53</double>
Entier :
<int>42</int>
Chaîne :
<string>Hello world!</string>
Structure :
<struct>
  <member>
    <name>test</name>
    <value><i4>1</i4></value>
  </member>
  <member>
    <name>test</name>
    <value><i4>2</i4></value>
  </member>
</struct>
```

Les appels de méthodes et la réponse ont également été traduits en XML.

Voici un exemple d'appel (`methodCall`) d'une méthode retournant le nom d'une ville en fonction d'un code postal :

```
<?xml version="1.0"?>
<methodCall> <methodName>exemples.getNomVille</methodName>
<params> <param> <value><i4>77000</i4></value> </param> </params>
</methodCall>
```

La réponse est alors :

```
<?xml version="1.0"?>
<methodResponse>
```

```
 <params>
    <param>
        <value><string>Melun</string></value>
    </param>
  </params>
</methodResponse>
```

Réaliser des échanges XML-RPC par programmation

Le site Apache, *http://ws.apache.org/xmlrpc/*, propose cette implémentation pour Java.

Voici un exemple d'échanges XML-RPC entre deux applications Java. On commence par écrire le code de traitement suivant :

```
import java.util.Calendar;
public class ServeurHorloge {
public int getHeure() {
 return Calendar.getInstance().get( Calendar.HOUR );
}
public int getMinute() {
 return Calendar.getInstance().get( Calendar.MINUTE );
}
public String getPays() { return "FRANCE"; }
}
```

Même si vous n'êtes pas familiarisé avec Java, on peut comprendre qu'il s'agit de trois méthodes, les deux premières retournant l'heure et les minutes en cours et la dernière retournant un pays.

Pour mettre à disposition ce code de traitement, il va falloir démarrer un serveur. Ce serveur nécessite un fichier de propriétés contenant un alias et la classe de traitement (dans notre exemple `ServeurHorloge`). Le serveur (ici jouant le rôle du squelette) utilisera la technologie Java `Reflection` pour réaliser les transferts d'appels de méthodes d'un client et l'acheminement des réponses (c'est un système d'invocation dynamique).

Voici un exemple de code serveur :

```
WebServer webServer = new WebServer(port);
XmlRpcServer xmlRpcServer = webServer.getXmlRpcServer();
PropertyHandlerMapping phm = new PropertyHandlerMapping();
URL u = ServeurRpc.class.getResource("MonService.properties");
phm.load(Thread.currentThread().getContextClassLoader(), u);
xmlRpcServer.setHandlerMapping(phm);
XmlRpcServerConfigImpl serverConfig = (XmlRpcServerConfigImpl) xmlRpcServer
.getConfig();
serverConfig.setEnabledForExtensions(true);
serverConfig.setContentLengthOptional(false);
System.out.println("Serveur demarre");
webServer.start();
```

La classe `WebServer` représente le serveur ; on indique le service que nous avons réalisé par la classe `PropertyHandlerMapping` et le fichier de propriétés `MonService.properties`. La méthode `start` met le serveur à l'écoute du port que nous avons spécifié dans la première ligne. Si vous souhaitez davantage de précision sur ces méthodes, la documentation de chaque classe est disponible à l'adresse : *http://ws.apache.org/xmlrpc/apidocs/index.html*.

Le fichier `MonService.properties` contient :

```
MonService.properties
Horloge=ServeurHorloge
```

Il reste à réaliser l'appel d'une méthode. Voici un exemple :

```
XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
// Le serveur XML-RPC
config.setServerURL(new URL("http://127.0.0.1:8080"));
XmlRpcClient client = new XmlRpcClient();
client.setConfig(config);
Object[] params = new Object[]{};
Integer result = (Integer) client.execute("Horloge.getHeure", params);
System.out.println( result );
```

La méthode `setServerURL` indique où se situe le serveur (ici nous avons fait un test local avec l'adresse IP 127.0.0.1). La classe `XmlRpcClient` sert à réaliser l'appel. Le service (`Horloge`) et la méthode (`getHeure`) sont précisés lors de l'appel de la méthode `execute`. La valeur de retour est ensuite convertie (opération appelée *cast*) selon le type attendu (ici un entier).

Les échanges avec SOAP

SOAP (*Simple Object Access Protocol*) est une recommandation du W3C (*http://www.w3.org/TR/SOAP/*), la dernière version étant la version 1.2 (*http://www.w3.org/TR/soap12/*). SOAP joue le même rôle que XML-RPC mais présente les avantages d'être standardisée par le W3C et de disposer d'une représentation des données moins verbeuse et plus fiable, grâce à l'usage des schémas W3C. SOAP s'appuie généralement sur HTTP (voire SMTP) pour la phase de transport des données, ce qui donne naissance aux services web.

Principal niveau de structure : l'enveloppe

Tout message SOAP est composé d'une enveloppe (`envelope`) contenant un en-tête optionnel (`header`) est un corps principal (`body`). On peut résumer cette structure par le code suivant :

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<soap:Header>
...
```

```
</soap:Header>
<soap:Body>
...
 <soap:Fault>
...
 </soap:Fault>
</soap:Body>
</soap:Envelope>
```

L'attribut `encodingStyle` indique quelle méthode d'encodage a été employée pour le message (toutefois, il y a peu de méthodes disponibles).

Première partie de l'enveloppe : l'en-tête

L'en-tête contient des informations contextuelles (transaction, authentification…). Un message SOAP peut circuler entre plusieurs intervenants. Les attributs `role` et `actor` servent à opérer un traitement de l'en-tête, uniquement sur certaines applications. L'attribut `mustUnderstand` peut rendre obligatoire le traitement de l'en-tête, ce qui a pour conséquence sa suppression, sauf si l'attribut `relay` a été associé avec la valeur `true`.

Voici un exemple d'en-tête :

```
<soap:Header>
<a:Authentication>
 <Username b:type='c:string'>user@example.org</Username>
 <MD5 b:type='c:string'>9b3e64e326537b4e8c0ff19e953f9673</MD5>
</a:Authentication>
</soap:Header>
```

Cet en-tête spécifie un compte d'accès au service (nom et mot de passe).

Deuxième partie de l'enveloppe : le corps

L'élément `body` va contenir l'appel de méthode et le résultat (éventuellement une erreur avec l'élément `Fault`).

Prenons un exemple de signature de méthode : `int calculer( int nombre )`. Cette méthode a donc un argument entier et retourne un résultat entier.

L'appel sera réalisé de cette façon :

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <soap:Envelope
   soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:soap ="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<soap:Body>
    <ns1:calculer
    xmlns:ns1="urn:MonService">
          <param1 xsi:type="xsd:int">123</param1>
```

```
    </ns1:doubleAnInteger>
</soap:Body>
</soap:Envelope>
```

Dans cet exemple, nous invoquons la méthode `calculer` avec la valeur `123` en argument. Notez bien que cette valeur est typée grâce au type simple `xsd:int`. L'élément `calculer` est lié à un espace de nom qui correspond en réalité au service traitant cette requête (similaire à ce que nous avons vu pour les échanges XML-RPC et le fichier de propriétés de l'implémentation Apache).

La réponse se présente sous cette forme :

```
<?xml version="1.0" encoding="UTF-8" ?>
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<soap:Body>
    <ns1:calculerResponse
    xmlns:ns1="urn:MonService"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
          <return xsi:type="xsd:int">246</return>
    </ns1:calculerResponse>
</soap:Body>
</soap:Envelope>
```

Le nom de la méthode est complété par le suffixe `Response`. L'élément `return` désigne la valeur résultat et est typé grâce à un type simple `xsd:int`.

Si le traitement de l'appel n'avait pas pu être correctement effectué, un code d'erreur (`code`) aurait été retourné par l'élément `Fault` avec un message explicite (`Reason`). Voici un exemple :

```
<soap:Body>
<soap:Fault>
   <soap:Code><soap:Value>soap:MustUnderstand</soap:Value></soap:Code>
<soap:Reason><soap:Text xml:lang='fr'>Une exception a été levée : …</soap:Text>
</soap:Reason>
</soap:Fault>
</soap:Body>
```

Dans cet exemple, le message d'erreur est indiqué en français grâce à la présence de l'attribut `xml:lang` et de la valeur `fr`.

Les échanges par les services web

Comme nous l'avons vu précédemment, SOAP est un moyen pour réaliser des appels de méthodes distantes. Cependant, il n'existe pas de mécanisme propre à SOAP pour découvrir les signatures des méthodes de chaque service. Ce système apparaît dans le contexte des services web grâce aux formats WSDL (*Web Services Description Language*) et UDDI (*Universal Description, Discovery and Integration*). Le premier décrit chaque méthode et la manière d'accéder au service (encodage des données, mode de transport…). Le deuxième joue le rôle d'annuaire de services web, un peu comme les pages jaunes.

Le format de description des services web : WSDL

Le format WSDL est standardisé par le W3C (*http://www.w3.org/TR/wsdl*). Il est composé de :

- Types de données : ils sont formalisés par l'élément `types` à l'aide d'un schéma W3C.
- Messages : chaque message caractérise, grâce à l'élément `message`, un accès à une méthode en entrée ou en sortie. L'élément `part` permet de décrire les arguments et la valeur de retour.
- Type de port : chaque type de port contient un ensemble d'opérations. Chaque opération, étant une association de messages en entrée et en sortie, est en quelque sorte un appel de méthode complet avec passage des paramètres et récupération d'une valeur. On utilise l'élément `portType` pour le caractériser.
- Liaisons : la liaison indique comment les opérations d'un type de port sont encodés et circulent. On utilise l'élément `binding`.
- Services : un service est un point d'accès à un ensemble de ports, chaque port étant relié à un élément de liaison et une adresse d'accès pour les clients.

Voici un exemple :

```
<?xml version="1.0"?>
<definitions name="cotation"
targetNamespace="http://exemple.com/cotation.wsdl"
 xmlns:tns="http://exemple.com/cotation.wsdl"
 xmlns:xsd1="http://exemple.com/cotation.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

   <types>
      <schema targetNamespace="http://example.com/cotation.xsd"
             xmlns="http://www.w3.org/2000/10/XMLSchema">
          <element name="demandeCotationAction">
             <complexType>
                 <sequence>
                     <element name="action" type="string"/>
                 </sequence>
             </complexType>
          </element>
          <element name="cotationAction">
             <complexType>
                 <sequence>
                     <element name="cotation" type="float"/>
                 </sequence>
             </complexType>
          </element>
      </schema>
   </types>

   <message name="demandeCotation">
```

```
        <part name="body" element="xsd1:demandeCotationAction"/>
    </message>

    <message name="resultatCotation">
        <part name="body" element="xsd1:CotationAction"/>
    </message>

    <portType name="actionPortType">
        <operation name="cotation">
           <input message="tns:demandeCotation"/>
           <output message="tns:resultatCotation"/>
        </operation>
    </portType>

    <binding
    name="cotationBinding"
     type="tns:actionPortType">
        <soap:binding
style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="cotation">
           <soap:operation soapAction="http://exemple.com/demandeCotation"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>

    <service name="actionService">
        <documentation>Cotation d'une action</documentation>
        <port name="actionPort" binding="tns:cotationBinding">
           <soap:address location="http://exemple.com/action"/>
        </port>
   </service>
</definitions>
```

Dans cet exemple, on commence par spécifier les types utilisés grâce à un schéma W3C. Ce schéma définit les éléments `demandeCotationAction` et `resultatCotation` dans l'espace de nom `http://example.com/cotation.xsd`. Toutes les références à ces éléments devront être qualifiées (pour lever toute ambiguïté dans le cas de déclarations multiples) d'où la déclaration d'un préfixe `xsd1`. Puis, nous avons deux messages, `demandeCotation` et `resultatCotation`, reliés aux éléments précédents. Comme nous avons un attribut `targetNamespace` à la racine, les définitions globales passent dans l'espace de nom associé et le type de port `actionPortType` doit donc faire des références aux messages précédents par un préfixe `tns`. C'est la même chose pour la liaison (`binding`), qui avec l'attribut `type` fait référence au type de port `actionPortType`. L'attribut `transport` indique par quel moyen les données circulent (HTTP, FTP, SMTP…) alors que l'attribut `style` indique comment les

messages sont exploités : soit sous la forme de document, soit sous une forme RPC (la différence est minime et concerne le format des messages). Chaque partie du message SOAP en entrée (input) et sortie (output) est ensuite définie. On indique comment sont structurés l'en-tête `header` (en option) et le corps `body`. L'attribut `use` peut prendre les valeurs `literal` ou `encoded` (en réalité, comme le style est `document`, seule la valeur `literal` a un sens). Cet attribut ne sert vraiment que pour un document de style RPC et inclut ou non des références au schéma dans les messages de requête et de réponse via l'attribut `xsi:type`. Il reste enfin à définir le service avec une adresse d'accès via SOAP `http://exemple.com/action`.

Les annuaires UDDI

UDDI (*Universal Description, Discovery and Integration*, *http://www.uddi.org/*) est une sorte de registre XML pour publier et utiliser des services web sur Internet. L'idée est que, puisque beaucoup de services sont utiles sur Internet, comme un service de vérification des numéros de carte de crédit, des sociétés pourraient proposer leur service à d'autres sociétés clientes.

UDDI comprend trois parties :

- Des pages blanches : informations sur les sociétés qui ont publié des services (adresse…).
- Des pages jaunes : il s'agit d'une répartition des services en catégories.
- Des pages vertes : elles regroupent des informations techniques et les services disponibles.

Le registre XML est interrogeable comme un service web classique, c'est-à-dire avec SOAP et un descripteur WSDL.

Voici quelques serveurs UDDI :

- jUDDI est une implémentation open source réalisée par le groupe Apache (*http://ws.apache.org/juddi/*).
- La solution Microsoft est décrite à cette adresse : *http://www.microsoft.com/windowsserver2003/technologies/idm/uddi/default.mspx*.
- IBM intègre avec WebSphere un serveur UDDI (*http://www-128.ibm.com/developerworks/websphere*).

Programmation des services web

Nous n'allons pas détailler chaque possibilité d'utilisation des services web, la plupart des langages proposant des API pour effectuer la création et l'appel d'un tel service. En complément, nous pouvons citer NuSoap pour les développeurs PHP (*http://sourceforge.net/projects/nusoap/*).

La technologie Apache : Axis

Axis est un projet Open Source réalisé par le groupe Apache (*http://ws.apache.org/axis/*) avec une implémentation Java et C++.

L'appel d'un service web

Commençons par regarder comme réaliser un appel d'un service web en Java. Voici un exemple :

```
String endpoint = "http://nagoya.apache.org:5049/axis/services/echo";
Service  service = new Service();
Call call    = (Call) service.createCall();
call.setTargetEndpointAddress( new java.net.URL(endpoint) );
call.setOperationName(new QName("http://soapinterop.org/", "echoString"));
String ret = (String) call.invoke( new Object[] { "Hello!" } );
System.out.println("Envoi 'Hello!', recu '" + ret + "'");
```

Cela diverge peu de ce que nous avions vu avec l'implémentation Apache de XML-RPC. La classe `Service` sert à construire un appel de méthode et à lire la réponse. Chaque méthode se traduit par un objet implémentant l'interface `Call` (c'est-à-dire respectant certaines méthodes). Pour plus d'informations, la documentation de chaque classe est disponible à l'adresse : *http://ws.apache.org/axis/java/apiDocs/index.html*. La méthode `setTargetEndPointAddress` correspond à l'URL du service web, tandis que la méthode `setOperationName` caractérise la méthode distante que l'on souhaite invoquer. Enfin, la méthode `invoke` effectue l'opération finale en créant une requête SOAP et en retournant l'objet résultat. Les arguments de la méthode `invoke` correspondent aux arguments de la méthode distante (ici `echoString`).

Voici les messages SOAP qui vont circuler lors de l'exemple précédent. Tout d'abord, la requête :

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:echoString xmlns:ns1="http://soapinterop.org/">
      <arg0 xsi:type="xsd:string">Hello!</arg0>
    </ns1:echoString>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Enfin, la réponse :

```
<?xml version="1.0" encoding="UTF-8"?>
 <SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
   <ns1:echoStringResponse xmlns:ns1="http://soapinterop.org/">
    <result xsi:type="xsd:string">Hello!</result>
```

```
    </ns1:echoStringResponse>
   </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

La création d'un service web

Nous avons au préalable installé Axis dans le moteur de servlet Tomcat (*http://tomcat.apache.org/*).

```
import java.util.Calendar;

public class MonService {
 public int getHeure() {
 return Calendar.getInstance().get( Calendar.HOUR );
 }
 public int getMinute() {
  return Calendar.getInstance().get( Calendar.MINUTE );
 }
 public String getPays() { return "FRANCE"; }
}
```

Nous avons repris la même classe que lors de notre exemple XML-RPC, c'est-à-dire des méthodes `getHeure` et `getMinute` retournant l'heure courante. Le fichier source est nommé `MonService.jws` et est copié dans le répertoire `axis` dans le contexte d'application du serveur d'application (pour Tomcat, le répertoire `webapps`). Au premier accès, un descripteur WSDL est créé à partir de la structure de notre classe. L'appel à ce service est alors possible comme pour n'importe quel service web.

Voici un extrait du descripteur WSDL produit et accessible via ce type d'URL (la machine peut varier ; ici il s'agit d'un accès local) : `http://localhost:8080/axis/MonService.jws?WSDL`.

```
<wsdl:definitions targetNamespace="http://localhost:8080/axis/MonService.jws">
<wsdl:message name="getPaysResponse">
<wsdl:part name="getPaysReturn" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="getPaysRequest">
</wsdl:message>
<wsdl:message name="getMinuteResponse">
<wsdl:part name="getMinuteReturn" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="getMinuteRequest">
</wsdl:message>
<wsdl:message name="getHeureResponse">
…
```

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class TestClient {
```

```
public static void main(String [] args) {
try {
 String endpoint ="http://127.0.0.1:8080/axis/MonService.jws";
  Service  service = new Service();
  Call     call    = (Call) service.createCall();
  call.setTargetEndpointAddress( new java.net.URL(endpoint) );
  call.setOperationName( "getHeure" );
  Integer ret = (Integer) call.invoke( new Object[] {} );
  System.out.println("Il est " + ret + " h" );
 } catch (Exception e) {
  System.err.println(e.toString());
 }
}
}
```

Dans cet exemple, similaire à notre premier cas d'appel, nous avons invoqué la méthode `getHeure` et avons affiché l'entier résultat (variable `ret`).

Le générateur de code

La commande `WSDL2Java` peut être utilisée afin de produire le code client à partir d'un descripteur WSDL. Ce code client joue le rôle de stub et garantit une transparence à l'appel du service.

Voici un exemple d'utilisation :

```
java org.apache.axis.wsdl.WSDL2Java -o c:/ c:/MonService.wsdl
```

Cette commande a produit quatre fichiers :

- `MonService.java` : l'interface (ensemble de signatures de méthodes sans code de traitement) avec les méthodes possibles.
- `MonServiceService.java` : une interface d'accès au service précédent.
- `MonServiceServiceLocator.java` : implémentation (c'est-à-dire que les méthodes contiennent du code) de l'interface précédente.
- `MonServiceSoapBindingStub.java` : contient toutes les opérations avec l'emploi de SOAP et implémente la première interface.

Voici ensuite comment on peut utiliser ces classes pour réaliser un appel :

```
MonServiceServiceLocator loc = new MonServiceServiceLocator();
MonService s = loc.getMonService();
System.out.println( s.getHeure() );
```

Cette solution est beaucoup plus simple que celle que nous avions employée au début, car tous les accès sont masqués dans des implémentations et l'on ne manipule plus que des interfaces.

La technologie Java JSE 6

Le JSE 6 (*Java Standard Edition* : *http://java.sun.com/javase/*) comporte des fonctionnalités pour créer et invoquer un service web. Ces fonctionnalités ont été très simplifiées grâce aux annotations. Ces dernières servent à ajouter des sortes de commentaires, ou instructions, dans un fichier source, qui peuvent ensuite être traités par un programme externe.

Voici un exemple de fichier source :

```
@WebService
public class ServiceSimple {
@WebMethod
public String echo( String p ) {
return p;
}
}
```

Cet exemple comporte deux annotations : `WebService` et `WebMethod`. Elles servent à associer la classe `ServiceSimple` à un service web et la méthode `echo` à une opération possible.

Nous utilisons ensuite la commande `wsgen`, pour générer les codes pour le serveur, de la façon suivante :

```
wsgen -classpath bin -d src ServiceSimple
```

Cela a pour conséquence de créer deux classes, `Echo` et `EchoResponse`, correspondant à l'utilisation de la méthode `echo`. Ces classes vont effectuer la conversion avec le format SOAP et seront utilisées indirectement par le code qui suit.

Le service web peut ensuite être activé grâce à ce code :

```
import javax.xml.ws.Endpoint;
public class Demarrer {
public static void main( String[] args ) {
    System.out.println( "Service Web demarre" );
    Endpoint.publish( "http://localhost:9090/echo", new ServiceSimple() );
}
}
```

La méthode `publish` possède en argument l'URL d'accès au service web et un objet d'implémentation. Il est difficile de faire plus simple.

Pour l'utilisation du service web, il suffit d'avoir un accès au descripteur WSDL et d'utiliser la commande `wsimport` de la façon suivante :

```
wsimport -d bin -s src -p client http://localhost:9090/echo?WSDL
```

Cela a pour conséquence de produire six classes :

- `Echo.java` : une requête du client vers la méthode `echo` (associée à une requête SOAP) ;
- `EchoResponse.java` : la réponse du client (associée à une réponse SOAP) ;
- `ObjectFactory.java` : classe utilitaire associée à JAXB (une solution réalisant une correspondance, on parle de *mapping*, entre un document XML et des objets java) ;

- `package-info.java` : association entre un espace de nom et un package ;
- `ServiceSimple.java` : interface avec les méthodes du service web ;
- `ServiceSimpleService.java` : classe utilitaire pour appeler le service web.

Il ne reste plus qu'à effectuer l'appel à notre service web par :

```
public class TestClient {
public static void main( String[] args ) {
    ServiceSimpleService ss = new ServiceSimpleService();
    ServiceSimple s = ss.getServiceSimplePort();
    System.out.println(  "Appel hello world --> " + s.echo( "hello world" ) );
}
}
```

La plate-forme .NET

La plate-forme .NET facilite aussi la création de service web. Voici un exemple :

```
<%@ WebService Language="VB" Class="TempConvert" %>

Imports System
Imports System.Web.Services

Public Class TempConvert :Inherits WebService

<WebMethod()> Public Function FahrenheitToCelsius
(ByVal Fahrenheit As Int16) As Int16
Dim celsius As Int16
 celsius = ((((Fahrenheit) - 32) / 9) * 5)
 Return celsius
End Function

<WebMethod()> Public Function CelsiusToFahrenheit
(ByVal Celsius As Int16) As Int16
Dim fahrenheit As Int16
 fahrenheit = ((((Celsius) * 9) / 5) + 32)
 Return fahrenheit
End Function
End Class
```

L'instruction `<WebMethod()>` indique les méthodes disponibles dans le service web. La classe `TempConvert` est un service web par héritage de la classe `WebService`. Notre classe doit être dans un fichier d'extension `asmx` et disponible dans le serveur web qui est alors opérationnel (construction des classes et du fichier WSDL au premier accès).

Les échanges XML avec Ajax

Nous avons également choisi de parler d'Ajax (*Asynchronous JavaScript and XML*) en termes d'échange XML. Ajax n'est pas vraiment une technologie, mais un ensemble de technologies servant à faire communiquer un navigateur et un serveur web sans effectuer de rechargement de la totalité de la page (chargement des données en tâche de fond).

La réponse du serveur peut être du texte simple ou un document XML. Tous les traitements sont réalisés en JavaScript à l'aide d'un objet `XMLHttpRequest`. Sans entrer dans le détail sur la manière d'utiliser Ajax en fonction des navigateurs, il faut comprendre qu'un objet `XMLHttpRequest` est associé à une méthode de traitement pour chaque réponse du serveur.

Voici un exemple simplifié de code JavaScript pour Internet Explorer :

```
var xmlHttp;
function initAjax() {
 xmlHttp = new ActiveXObject( "Msxml2.XMLHTTP" );
 xmlHttp.onreadystatechange = traiterReponse;
 xmlHttp.open( "GET", "http://localhost:8084/bourse/AJAXServlet?nom="
+ document.f.nom.value, true );
 xmlHttp.send( null );
 return false;
}
function traiterReponse() {
 if ( xmlHttp.readyState == 4 ) {
  if ( xmlHttp.status == 200 ) {
   var doc = xmlHttp.responseXML;
   alert( doc.firstChild.firstChild.nodeValue ); }
}
}
```

La fonction `initAjax` crée un objet de requête (`XMLHttpRequest`) lié à la variable `xmlHttp`. L'attribut `onreadystatechange` désigne la fonction `traiterReponse` chargée de gérer la réponse du serveur. Les instructions `open` et `send` servent à réaliser l'envoi d'une requête au serveur. Lorsque la réponse du serveur est obtenue et complète, les flags `readyState` et `status` possèdent respectivement les valeurs `4` et `200`. Le champ `responseXML` contient alors une réponse XML sous forme de référence DOM à l'arbre résultat. Nous aborderons l'API DOM dans la partie programmation. Retenez pour l'instant que nous avons affiché, avec la commande `alert`, le texte présent dans le premier élément fils de la racine.

Côté serveur, il suffit de renvoyer un document XML (avec PHP, ASP, JSP…). Voici un exemple avec une simple servlet Java, qui génère un document XML avec une racine `status` et du texte. :

```
public class AJAXServlet extends HttpServlet {
protected void processRequest(
HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
 response.setContentType( "text/plain;charset=UTF-8" );
 PrintWriter out = response.getWriter();
 if ( "dupont".equals( request.getParameter( "nom" ) ) )
  out.write( "<status>ok</status>" );
 else
  out.write( "<status>accés refusé pour " + request.getParameter( "nom" )
  ➥+ "</status>"); out.close();
}
}
```

7

Les bases de données

Les bases de données ont un poids important dans la plupart des architectures. Une base de données optimise le stockage et les opérations de recherche en respectant une structure (schéma). Les bases de données relationnelles sont présentes partout et elles proposent de plus en plus de services complémentaires autour de XML. Les bases de données dites natives XML n'ont pas encore vraiment percé et il est difficile aujourd'hui de voir dans ces dernières un reliquat ou non des bases de données objet, qui sont restées confinées à un marché de niche.

Son rôle

Il est inutile de rappeler le rôle d'une base de données. Il semble plus intéressant d'aborder la relation entre un document XML et une base de données. On pourrait déjà considérer qu'un document XML est une forme de base de données, puisqu'il intègre des données structurées. La recherche dans ce document peut s'effectuer par des requêtes XPath ou XQuery.

Mais la comparaison s'arrête là : une base de données doit être capable de traiter de grands volumes et elle dispose de mécanismes d'indexation pour optimiser les recherches. Les bases de données travaillent en lecture et écriture et disposent généralement de mécanismes transactionnels pour éviter les incohérences d'état lors d'enchaînement d'opérations. Ces notions n'ont pas vraiment de sens dans un document XML, puisqu'il reste avant tout un fichier texte. Utiliser des fichiers XML de plusieurs giga-octets ne serait pas non plus envisageable pour des questions de performances.

Autre problème : un document XML est une arborescence de profondeur indéfinie, alors qu'avec les bases de données relationnelles l'information est divisée en tables, avec des relations entre les tables par clés primaires et clés étrangères. Cela implique donc que la correspondance entre les deux structures n'est pas directe : stocker un document XML dans une base de données relationnelle reviendra à ajouter une notion de parent et enfant sous forme de champs, ce qui n'est pas très performant pour l'écriture et la lecture et, pire, rendra la recherche très peu pratique.

D'une manière informelle, on pourrait définir une règle de structuration de ce type : pour chaque élément XML, créer une table avec une clé primaire (de type numérotation) et autant de champs que d'attributs ; pour chaque relation entre éléments parent et éléments enfant, ajouter également des champs clé secondaire. Pour réaliser une requête XPath, il faudra alors reconstruire le document XML par une routine parcourant les différentes tables. On le voit, si nous arrivons à bénéficier des facilités de stockage de la base, les performances de requête risquent d'être catastrophiques. Le problème sera le même en cas de mise à jour.

Les constructeurs de bases de données proposent de plus en plus des champs typés XML : le document XML devient alors une donnée comme une autre. Des requêtes avec XPath/XQuery sont mises à disposition pour ce type de champ.

La dernière solution est une base de données dite native XML. Cette base donne l'illusion de conserver le document XML tel quel, toutes les opérations habituelles sur les documents XML étant alors possibles. Cela peut être une solution élégante, mais il reste à savoir si ce type de base répond aux problématiques transactionnelles de volumétrie et de temps de réponse. Des *benchmarks* sont disponibles sur le site de Ronald Bourret (*http://www.rpbourret.com/*), mais étrangement, peu de résultats sont disponibles. Le problème est que les performances vont dépendre de la complexité de vos documents et de leur taille moyenne, et que cela nécessitera la mise en place d'une procédure de validation d'une plate-forme avec des tests en charge.

Quelques bases de données relationnelles

Nous allons étudier quelques usages possibles de XML avec certaines bases de données relationnelles.

La base MySQL

MySQL (*http://www.mysql.com/*) est incontournable, car c'est probablement la base de données (Open Source) la plus utilisée dans l'univers du Web. Dans un cadre plus large, cette base de données est classée, selon les études, entre la deuxième et troisième place en termes d'usage, derrière SQL Server et Oracle.

Cette base de données n'offre pour ainsi dire aucun service autour de XML. Il revient au développeur de réaliser la transformation d'un résultat de requête SQL en XML et, inversement, de stocker un document XML sous un ensemble de requêtes SQL de mise à jour.

On le comprendra facilement : la transformation en XML ne posera sans doute pas de problème pour une structure relativement peu profonde, tout comme le stockage sous la forme de requêtes SQL dans le cas d'une structure complexe.

Voici un exemple d'utilisation, tout d'abord pour générer un document XML. Nous sommes partis d'une table simple, `carnet`, comportant les champs `id`, `nom` et `prenom`. Pour cet exemple, nous avons utilisé Java et JDBC (*Java Database Connectivity*), une technologie associée qui s'appuie sur un mécanisme de pilotes, afin d'accéder à une base de données relationnelle :

```
static void convertisseurResultSetToXML(
StringBuffer sb, ResultSet rs ) throws SQLException {
sb.append( "<carnet>" );
while ( rs.next() ) {
 sb.append( "<personne id='" ).append( rs.getString( "id" )  ).append( "'>" );
 sb.append( "<nom>" ).append( rs.getString( "nom" ) ).append( "</nom>" );
 sb.append( "<prenom>" ).append( rs.getString( "prenom" ) ).append( "</prenom>" );
 sb.append( "</personne>" );
}
sb.append( "</carnet>" );
}

public static void main(String[] args) throws Throwable {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection c = DriverManager.getConnection( "jdbc:odbc:carnet"  );
 Statement st = c.createStatement();
 ResultSet set = st.executeQuery( "select * from carnet" );
 StringBuffer sb = new StringBuffer();
 convertisseurResultSetToXML( sb, set );
 System.out.println( sb );
 c.close();
}
```

La méthode `main` est le premier bloc d'instructions invoquée pour ce test. Sans entrer trop dans les détails propres à Java, notre code charge un pilote ODBC (*Open Database Connectivity*), via l'instruction `Class.forName`, puis établit une connexion à la base d'alias `carnet`. Une requête SQL est ensuite exécutée avec l'instruction `executeQuery`. La fonction `convertisseurResultSetToXML` se charge alors de lire chaque ligne du résultat et de produire le document XML correspondant (via le paramètre `sb`). Notre manière de procéder est bien sûr rudimentaire : on a supposé que les champs ne contenaient pas de caractères spéciaux (comme < …).

Nous aurions pu également passer par un arbre DOM placé en mémoire (consulter le prochain chapitre consacré à la programmation pour plus de détails), avec l'avantage de pouvoir réaliser facilement une transformation XSLT ou des requêtes XPath. Voici le même exemple avec une arborescence DOM :

```
static void convertisseurResultSetToXML( Document d, ResultSet rs ) throws SQLException {
Element e = d.createElement( "carnet" );
while ( rs.next() ) {
  Element pe = d.createElement( "personne" );
```

```
  pe.setAttribute( "id", rs.getString( "id" ) );
  Element pn = d.createElement( "nom" );
  Text tn = d.createTextNode( rs.getString( "nom" ) );
  pn.appendChild( tn );
  Element pp = d.createElement( "prenom" );
  Text tp = d.createTextNode( rs.getString( "prenom" ) );
  pp.appendChild( tp );
  pe.appendChild( pn );
  pe.appendChild( pp );
  e.appendChild( pe );
}
d.appendChild( e );
}

public static void main(String[] args) throws Throwable {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection c = DriverManager.getConnection( "jdbc:odbc:carnet" );
 Statement st = c.createStatement();
 ResultSet set = st.executeQuery( "select * from carnet" );
 Document d = DocumentBuilderFactory.newInstance().newDocumentBuilder()
 ➥.newDocument();
 convertisseurResultSetToXML( d, set );
 c.close();
}
```

Cette arborescence est associée au paramètre `d`. À chaque itération de la réponse SQL, un nouvel élément `personne` est construit puis complété par les éléments `nom` et `prenom`.

Voici un autre exemple similaire écrit en Perl, utilisant le module `DBI` :

```
use strict;
use DBI;

my $dbh = DBI->connect ("DBI:mysql:carnet",
                        "testuser", "testpass",
                        { RaiseError => 1, PrintError => 0});
my $sth = $dbh->prepare ("SELECT * FROM carnet");
$sth->execute ();
print "<?xml version=\"1.0\"?>\n";
print "<carnet>\n";
while (my ($nom, $prenom) = $sth->fetchrow_array ()) {
 print " <personne>\n";
 print "  <nom>$nom</nom>\n";
 print "  <prenom>$prenom</prenom>\n";
 print " </personne>\n";
}
$dbh->disconnect ();
print "</carnet>\n";
```

Prenons maintenant l'opération inverse, c'est-à-dire intégrer le contenu d'un document XML dans cette base de données. Nous sommes obligés d'utiliser un parseur XML pour

réaliser cela. Comme nous devons effectuer un parcours de l'arborescence, nous nous sommes appuyés sur un arbre DOM :

```
static void convertisseurXMLBD( Document d, PreparedStatement ps ) throws SQLException
{
 NodeList nl = d.getElementsByTagName( "personne" );
 for ( int i = 0; i < nl.getLength(); i++ ) {
  Element pe = ( Element )nl.item( i );
  String id = pe.getAttribute( "id" );
  Element nome = ( Element )pe.getElementsByTagName( "nom" ).item( 0 );
  Element prenome = ( Element )pe.getElementsByTagName( "prenom" ).item( 0 );
  String nom = nome.getFirstChild().getNodeValue();
  String prenom = prenome.getFirstChild().getNodeValue();
  ps.setString( 1, nom );ps.setString( 2, prenom );ps.setString( 3, id );
  ps.addBatch();
 }
 ps.executeBatch();
}
public static void main(String[] args) throws Throwable {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 Connection c = DriverManager.getConnection( "jdbc:odbc:carnet" );
 PreparedStatement st = c.prepareStatement( "insert into carnet values(?,?,?)");
 DocumentBuilderFactory factory1 = DocumentBuilderFactory.newInstance();
 DocumentBuilder db = factory1.newDocumentBuilder();
 Document doc = db.parse( "carnet.xml" );
 convertisseurXMLBD( doc, st );
 c.close();
}
```

Nous utilisons une requête d'insertion (variable `st`) qui contient des paramètres de requêtes. La fonction `convertisseurXMLBD` parcourt tous les éléments `personne` du document XML et associe à chaque valeur obtenue un paramètre de requêtes SQL (fonction `setString`). Lorsque la requête est prête (elle contient les données), la fonction `addBatch` sert à créer une nouvelle requête tout en stockant l'ancienne. Enfin, l'instruction `executeBatch` se charge de l'exécution de toutes les requêtes.

La base Oracle avec XSQL Servlet

XSQL Servlet est une technologie mise à disposition par Oracle (*http://www.oracle.com/technology/tech/xml/xdkhome.html*). Elle vise à automatiser une réponse XML liée à une requête SQL. Cette réponse peut ensuite être transformée par XSLT pour être présentée à un utilisateur. Cette solution présente les avantages d'être directement exploitable par configuration et de ne nécessiter aucune programmation, en dehors de la préparation d'un document XSLT.

Cette solution s'appuie sur la plate-forme JEE (*Java Enterprise Edition*) et est donc multi-plates-formes. Elle bénéficie aussi de toute la richesse Java en termes de pilotes de bases de données. Nous avons réalisé un test avec une base MySQL. Attention, il vous faut, outre une machine virtuelle Java (*Javal Virtual Machine*, ou JVM) au moins en version 1.4, installer un moteur de servlet de style Tomcat (*http://tomcat.apache.org/*).

Une fois la servlet installée, vous disposez du fichier de configuration `WEB-INF/classes/XSQLConfig.xml`, dont voici un extrait :

```
<connectiondefs>
    <connection name="employe">
      <username>root</username>
      <password></password>
      <dburl>jdbc:odbc:entreprise</dburl>
      <driver>sun.jdbc.odbc.JdbcOdbcDriver</driver>
      <autocommit>false</autocommit>
    </connection>
    <connection name="demo">
      <username>scott</username>
      <password>tiger</password>
      <dburl>jdbc:oracle:thin:@localhost:1521:ORCL</dburl>
      <driver>oracle.jdbc.driver.OracleDriver</driver>
      <autocommit>false</autocommit>
    </connection>
...
```

Chaque élément `connection` correspond à une configuration de connexion à une base de données. L'attribut `name` est important pour pouvoir s'y référer lors d'une requête. On trouvera des informations classiques de connexion, comme le compte et mot de passe, ainsi que l'URL d'accès ; à noter aussi la présence du pilote JDBC avec l'élément `driver` (dans notre cas un accès par ODBC).

Nous avons ajouté la connexion `employe` qui pointe vers une base MySQL dont l'alias ODBC est `entreprise`.

On réalise ensuite un fichier d'extension `xsql` pour chaque requête. Ce dernier contient une requête SQL. Voici un premier exemple :

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql" connection="employe">
<xsql:query>
    show database;
</xsql:query>
</page>
```

L'élément `page` indique la connexion sur laquelle nous travaillons : nous retrouvons donc la valeur `employe` associée à l'attribut `connection`. L'élément `query` est présent pour chaque requête SQL qu'on souhaite réaliser. Dans cet exemple, nous demandons l'ensemble des tables disponibles.

La réponse se présente sous cette forme :

```
<?xml version = '1.0'?>
<page>
<ROWSET>
<ROW num="1"><Database>information_schema</Database></ROW>
<ROW num="2"><Database>entreprise</Database></ROW>
<ROW num="3"><Database>mysql</Database></ROW>
```

```
<ROW num="4"><Database>test</Database></ROW>
</ROWSET>
</page>
```

Il y a autant d'éléments ROW que de lignes de réponse ; un attribut `num` contient le numéro de réponse (information utile si plusieurs requêtes sont effectuées).

Prenons maintenant un exemple plus sophistiqué. Notre base contient une table `employe` comportant les champs `nom`, `prénom`, `date de naissance` et `entreprise`.

Nous avons créé le fichier requête `test.xsql` suivant :

```
<?xml version="1.0"?>
<page xmlns:xsql="urn:oracle-xsql" connection="employe">
<xsql:query rowset-element="groupe" row-element="employe">
 select * from employe;
</xsql:query>
</page>
```

Cette requête liste les différents enregistrements de la table `employe`. Les attributs `rowset-element` et `row-element` servent à changer le nom de l'élément pour l'ensemble de la réponse et le nom de l'élément pour chaque ligne de réponse. Voici le résultat en utilisant cette URL : `http://127.0.0.1:8080/xsql/test.xsql` :

```
<?xml version = '1.0'?>
<page>
<groupe>
 <employe num="1">
  <nom>Martin</nom>
  <prenom>Jean</prenom>
  <dateNaissance>1970-05-13</dateNaissance>
  <entreprise>JAPISOFT</entreprise>
 </employe>
...
</groupe>
</page>
```

Comme vous pouvez le constater, chaque champ est transformé en élément de même nom. La dernière étape consiste maintenant à réaliser une feuille de styles et effectuer un affichage dans un navigateur via le fichier `employe.xslt` suivant :

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
 <html>
  <body><xsl:apply-templates select="page/groupe"/>
  </body>
 </html>
</xsl:template>
<xsl:template match="/page/groupe">
 <xsl:for-each select="employe">
  <p>
```

```
  <ul>
   <li><xsl:value-of select="nom"/></li>
   <li><xsl:value-of select="prenom"/></li>
   <li><xsl:value-of select="dateNaissance"/></li>
  </ul>
  </p>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Ce document XSLT (voir le chapitre 5, consacré à la publication) effectue un affichage de chaque employé sous la forme d'un triplet nom, prénom et date de naissance.

Il ne nous reste plus qu'à utiliser un fichier de requête avec un lien vers notre feuille de styles, selon cette méthode :

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="employe.xslt"?>
<page xmlns:xsql="urn:oracle-xsql" connection="employe">
<xsql:query rowset-element="groupe" row-element="employe">
    select * from employe;
</xsql:query>
</page>
```

Nous obtenons alors, au prochain accès à notre fichier XSQL, une page HTML.

Quelques bases de données natives XML

Après avoir présenté quelques usages des bases de données relationnelles, nous pouvons maintenant aborder d'autres formes de bases, probablement plus en adéquation avec ce que nous souhaitons faire avec des documents XML.

La base Open Source Xindice

Xindice est une base de données Open Source native XML réalisée par le groupe Apache (*http://xml.apache.org/xindice/*). Elle peut fonctionner via un moteur de servlet, type Tomcat (*http://tomcat.apache.org/*) et est donc multi-plates-formes. Elle autorise l'ajout et la suppression de document XML, ainsi qu'une recherche par requête XPath 1.0. La communication avec la base de données se fait par le format XML-RPC (voir le chapitre 6) ce qui autorise un accès via d'autres langages de programmation.

Interactions avec Xindice

Les documents XML sont répartis en collection. Des instructions en ligne de commandes sont disponibles, en voici un échantillon :

```
Commande pour lister les collections :
xindice list_collections -c xmldb:xindice://localhost:8080/db
Ajouter une collection test :
```

```
xindice add_collection -c xmldb:xindice://localhost:8080/db -n test
Ajouter un document livre1.xml dans la collection test, livre1 correspond à un alias
➥(sorte d'identifiant du document) :
xindice add_document -c xmldb:xindice://localhost:8080/db/test -f livre1.xml -n livre1
Retrouver un document à partir de l'alias livre1. Le document final est nommé
➥livre1b.xml :
xindice retrieve_document -c xmldb:xindice://localhost:8080/db/test -n livre1 -f livre1b.xml
Effacer un document selon un alias :
xindice delete_document -c xmldb:xindice://localhost:8080/db/test -n livre1
Requête XPath pour trouver tous les documents dont la racine livre possède un
➥attribut titre contenant le mot livre :
xindice xpath_query -c xmldb:xindice://localhost:8080/db/test -q
➥"/livre[contains(@titre, 'livre')]"
<livre src:col="/db/test" src:key="livre1" titre="Mon livre"
➥xmlns:src="http:ml.apache.org/xindice/Query">
..
</livre>
<livre src:col="/db/test" src:key="livre2" titre="Mon livre"
➥xmlns:src="http://xml.apache.org/xindice/Query">
...
</livre>
```

Dans chacune de ces instructions, le paramètre –c désigne la localisation de la base et, éventuellement, de la collection. Chaque document réponse (lié à une requête XPath) est complété par les attributs `col` et `key`, représentant respectivement le chemin de la collection et l'alias du document.

Exercice 1

Quelques manipulations

Il vous est demandé de réaliser les opérations suivantes :

L'installation de la base de données (packages à récupérer : *http://xml.apache.org/xindice/download.cgi* et l'installation de Tomcat : *http://tomcat.apache.org/*).

Votre service informatique vous envoie des documents XML correspondant à un résultat de formulaire de saisie. Ces formulaires contiennent des informations sur les adhérents d'un club de sport, c'est-à-dire le nom, le prénom, l'âge, le numéro de téléphone et les activités souhaitées.

Imaginez le format XML associé.

Faites plusieurs ajouts de documents dans la base de données. Comptez ensuite le nombre d'inscrits par activité.

Programmation avec Xindice

Cette base de données est dotée d'une API de programmation en Java (*http://xml.apache.org/xindice/guide-developer.html*). Nous allons en aborder ici quelques aspects.

L'insertion d'un document XML

Voici un exemple d'insertion d'un document XML, carnet.xml, dans la collection carnets :

```
Collection col = null;
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);

Database database = ( Database )c.newInstance();
DatabaseManager.registerDatabase( database );
col = DatabaseManager.getCollection("xmldb:xindice://127.0.0.1:8080/db/carnets");

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setIgnoringElementContentWhitespace( true );
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse( "carnet.xml" );

XMLResource document = (XMLResource) col.createResource( "carnet1", "XMLResource");
document.setContentAsDOM( d );
col.storeResource( document );
```

Le principe retenu consiste à obtenir un objet `col`, représentant la collection, par l'instruction `getCollection`. Cet objet dispose d'une méthode `storeResource` qui accepte un arbre DOM résultat du *parsing* du fichier `carnet.xml`.

La récupération d'un document XML

```
Collection col = null;
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);

Database database = (Database) c.newInstance();
DatabaseManager.registerDatabase(database);
col = DatabaseManager.getCollection("xmldb:xindice://127.0.0.1:8080/db/carnets");

XMLResource res = ( XMLResource )col.getResource( "carnet1" );
System.out.println( res.getContent() );
```

Ce cas est similaire au précédent, mais en employant l'instruction `getResource` avec l'alias d'un document de la collection. On récupère alors le document sous une forme texte par l'instruction `getContent`.

Le requêtage de documents XML

```
Collection col = null;
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);

Database database = (Database) c.newInstance();
DatabaseManager.registerDatabase(database);
col = DatabaseManager.getCollection("xmldb:xindice://127.0.0.1:8080/db/carnets");

String xpath = "/carnet/personne[@id='p1']";
XPathQueryService service =
```

```
(XPathQueryService) col.getService("XPathQueryService", "1.0");
ResourceSet resultSet = service.query(xpath);
ResourceIterator results = resultSet.getIterator();
while (results.hasMoreResources()) {
Resource res = results.nextResource();
System.out.println((String) res.getContent());
}
```

Nous avons effectué une requête XPath filtrant tous les documents possédant un élément `personne` et portant un attribut `id` de valeur `p1`. Cette requête fait appel à un service que l'on obtient auprès de la collection avec la méthode `getService`. L'instruction `query` déclenche alors l'exécution de la requête XPath et retourne l'ensemble des nœuds résultats. On parcourt cet ensemble via l'instruction `nextResource` ; l'instruction `getContent` nous donne, au fur et à mesure, le texte XML correspondant.

Voici le document obtenu :

```
<personne id="p1" src:col="/db/carnets" src:key="carnet1"
 xmlns:src="http://xml.apache.org/xindice/Query"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<nom>Fog</nom>
<prenom>Phileas</prenom>
</personne>
```

La base Open Source Berkeley DB XML

Berkeley DB est un ensemble de bases de données Open Source. Oracle propose une version native XML (*http://www.oracle.com/database/berkeley-db/index.html*) que nous allons étudier. Cette base est capable de gérer de gros volumes de données (256 To) dans un contexte transactionnel.

Interaction avec la base

Les documents XML sont manipulables en ligne de commandes ou par programmation. Les requêtes peuvent être écrites en XPath 2.0/XQuery.

Les documents sont stockés dans des conteneurs (*containers*).

Le *shell* pour envoyer des commandes s'appelle `dbxml`. Nous nous retrouvons alors avec une console (comme `command.com`/`cmd.exe` sous Windows, ou une console Unix/Linux).

L'ajout de documents XML dans un conteneur

Commençons par créer un conteneur pour nos documents :

```
dbxml> createContainer test.dbxml
Creating node storage container with nodes indexed
```

Le conteneur est associé à un fichier de stockage (ici `test.dbxml`) qu'il ne faut jamais éditer directement.

L'ajout de document peut être réalisé de différentes façons, mais passe, dans tous les cas, par la déclaration d'un alias (premier argument) avec la commande `putDocument`. La première solution, peu pratique, consiste à insérer directement le document en terminant la commande par `s`.

```
dbxml> putDocument carnet1 '<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
</carnet>' s
Document added, name = carnet1
```

La deuxième solution utilise un répertoire fichier (c'est le *flag* `f`, à la fin de l'instruction, qui sert à indiquer qu'il s'agit bien d'un path) :

```
dbxml> putDocument carnet2 c:/carnet2.xml f
Document added, name = carnet2
```

À noter qu'un document inséré peut être retrouvé par la commande `getDocuments` suivi de l'alias.

La construction du conteneur peut également être associée à un processus de validation à l'aide de schéma W3C :

```
dbxml> createContainer test2.dbxml d validate
Creating document storage container, with validation
```

Ainsi les documents associés à `test2.dbxml` seront toujours validés avant insertion.

Le requêtage de documents XML

Nous pouvons maintenant effectuer des requêtes XQuery dans la collection. Par exemple :

```
dbxml> query collection("test.dbxml")/carnet
2 objects returned for eager expression 'collection("test.dbxml")/carnet'
dbxml> print
<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
</carnet>
<carnet>
        <personne nom="brillant" prenom="alex"/>
        <personne nom="briand" prenom="aristide"/>
</carnet>
```

Dans ce test, nous avons indiqué le conteneur utilisé par la fonction `collection` suivie de notre requête : ici, nous récupérons tous les documents de racine `carnet`.

Voici une autre requête plus sophistiquée, qui filtre tous les noms commençant par `dup`.

```
dbxml> query collection("test.dbxml")/carnet/personne[starts-with(@nom,'dup')]
2 objects returned for eager expression 'collection("test.dbxml")/carnet/personn
e[starts-with(@nom,'dup')]'
dbxml> print
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
```

L'optimisation des performances par les index

Lors d'une requête, tous les documents du conteneur sont analysés, ce qui peut prendre beaucoup de temps. Pour améliorer les performances, des index peuvent être positionnés. Ils se composent d'un nœud et d'une stratégie (type d'index).

Voici les différents types d'index à disposition :

- `none-none-none` : désactiver l'index ;
- `node-element-presence` : présence d'un élément ;
- `node-attribute-presence` : présence d'un attribut ;
- `node-element-equality-string` : test d'égalité de chaîne sur le texte d'un élément ;
- `node-element-equality-number` : test d'égalité de nombre sur le contenu d'un élément ;
- `node-element-substring-string` : test de sous-chaîne sur le texte d'un élément ;
- `node-attribute-equality-string` : test d'égalité sur un attribut ;
- `node-attribute-equality-number` : test d'égalité de nombre sur un attribut ;
- `node-attribute-substring-string` : test de sous-chaîne sur un attribut.

Voici un exemple, en rapport avec la dernière requête que nous avons effectuée :

```
dbxml> addIndex "" personne/@nom node-attribute-substring-string
Adding index type: node-attribute-substring-string to node: {}:personne/@nom
```

Les recherches croisées

Il est également possible de travailler avec plusieurs conteneurs.

Construisons, tout d'abord, un nouveau conteneur, `tel.dbxml` :

```
dbxml> createContainer tel.dbxml
Creating node storage container with nodes indexed
```

Ajoutons également plusieurs documents :

```
dbxml> putdocument tel1 '<tels><tel nom="dupont" prenom="jean" val="12345678"/>
</tels>' s
Document added, name = tel1

dbxml> putdocument tel2 '<tels><tel nom="dupond" prenom="alex" val="987654321"/>
</tels>' s
Document added, name = tel2
```

Les deux conteneurs que nous avons créés peuvent être chargés par la commande `preload` :

```
dbxml> preload test.dbxml
dbxml> preload tel.dbxml
```

Nous pouvons maintenant effectuer une requête XQuery croisant les deux conteneurs :

```
dbxml> query 'for $p in collection( "test.dbxml" )//personne
for $t in collection( "tel.dbxml" )//tel[ @nom = $p/@nom ]
```

```
return $t'
2 objects returned for eager expression 'for $p in collection( "test.dbxml" )//personne
dbxml> print
<tel nom="dupont" prenom="jean" val="12345678"/>
<tel nom="dupond" prenom="alex" val="987654321"/>
```

Nous avons affiché les noms et les numéros de téléphone communs entre les collections `test.dbxml` et `tel.dbxml`.

La recherche spécifique à un document

La fonction `doc` peut être utilisée pour effectuer une requête sur un document spécifique. Elle fonctionne sous forme de path avec la collection et l'alias du document XML (celui choisi lors de l'ajout du document).

Exemple :

```
dbxml> query 'doc("test.dbxml/carnet1")//*'
3 objects returned for eager expression 'doc("test.dbxml/carnet1")//*'
dbxml> print
<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
</carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
```

L'ajout complémentaire de métadonnées avec les documents XML

À un document XML, il est possible d'associer des métadonnées. Ces dernières ne font pas vraiment partie du document, mais peuvent contenir toute information que l'on souhaite y associer, comme la date de modification, le nom du concepteur, la dernière personne à l'avoir modifié… Il est ensuite possible de réaliser des requêtes prenant en compte ces valeurs par la fonction `dbxml:metadata`.

Exemple avec la fonction `setMetaData` :

```
dbxml> setMetaData carnet1 '' auteur string alex
MetaData item 'auteur' added to document carnet1
dbxml> setMetaData carnet2 '' auteur string alex
MetaData item 'auteur' added to document carnet2
```

Ici, nous avons associé aux documents `carnet1` et `carnet2` un champ `auteur` contenant une chaîne (`string`) de valeur `alex`.

Voici maintenant un exemple de requête filtrant tous les documents pour un auteur donné :

```
dbxml> query 'collection("test.dbxml")/carnet[dbxml:metadata("auteur")="alex"]
'
2 objects returned for eager expression 'collection("test.dbxml")
➥/carnet[dbxml:metadata("auteur")="alex"]
'
```

```
dbxml> print
<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
</carnet>
<carnet>
        <personne nom="brillant" prenom="alex"/>
        <personne nom="briand" prenom="aristide"/>
</carnet>
dbxml>
```

La modification d'un document XML

La modification d'un document va être effectuée en deux temps. En premier lieu, on sélectionne un ensemble de documents par une requête XQuery. Puis, on applique une commande de modification en indiquant la partie que l'on souhaite modifier et la nouvelle valeur (élément, attribut…).

Les principales commandes sont :

- `append` : ajout d'un élément ou attribut ;
- `insertAfter/insertBefore` : ajout d'un élément avant ou après un autre élément ;
- `removeNodes` : suppression d'un élément ou attribut ;
- `renameNodes` : renommage d'un élément ou attribut.

Exemple de modification :

```
dbxml> query 'doc("test.dbxml/carnet1")'
1 objects returned for eager expression 'doc("test.dbxml/carnet1")'
dbxml> append ./carnet element personne ''
Appending into nodes: ./carnet an object of type: element with name: personne and content:
1 modifications made.

dbxml> print
<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
<personne/></carnet>

dbxml> append ./carnet/personne[last()] attribute nom 'dupons'
Appending into nodes: ./carnet/personne[last()] an object of type: attribute with name:
➥nom and content: dupons
1 modifications made.

dbxml> append ./carnet/personne[last()] attribute prenom 'test'
Appending into nodes: ./carnet/personne[last()] an object of type: attribute with name:
➥prenom and content: test
1 modifications made.

dbxml> print
```

```
<carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
<personne nom="dupons" prenom="test"/></carnet>
dbxml>
```

Dans cet exemple, nous avons ajouté un élément `personne` dans le document XML d'alias `carnet1` ; nous avons ensuite ajouté un attribut `nom` et un attribut `prenom` à cet élément (qui se trouve être le dernier, d'où l'utilisation de la fonction `last`).

Activation de la validation dans un conteneur

La validation commence par la création d'un conteneur validant automatiquement les documents insérés (voir la section précédente : *L'ajout de documents XML dans un conteneur*).

Nous avons utilisé le schéma suivant, `carnet.xsd` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="carnet">
  <xs:complexType>
   <xs:sequence maxOccurs="unbounded">
    <xs:element name="personne">
     <xs:complexType>
      <xs:attribute name="nom" type="xs:string"/>
      <xs:attribute name="prenom" type="xs:string"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Il reste à ajouter des documents XML validés suivant ce schéma :

```
dbxml>putDocument carnet3 '<carnet xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="carnet.xsd">
<personne nom="aa" prenom="bb"/>
</carnet>' s
Indexer    - test2.dbxml - add unique-node-metadata-equality-string, key={Size=9
 Hex=5a016361726e657433}, data={Size=3 Hex=000200}
Document added, name = carnet3

dbxml>putDocument carnet4 '<carnet xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="carnet.xsd">
<personne nom="cc" preno="dd"/>
</carnet>' s
stdin:147: putDocument failed, Error: XML Indexer:  Parse error in document at l
ine, 2, char 32. Parser message: Attribute 'preno' is not declared for element '
personne'
dbxml>
```

Nous avons souhaité ajouter deux documents d'alias, respectivement `carnet3` et `carnet4`. Dans les deux cas, nous avons associé le document au schéma `carnet.xsd`. Le premier cas est correct, alors que le deuxième ne l'est pas à cause de l'attribut `preno` utilisé à la place de `prenom` il n'est donc alors pas ajouté dans la base.

Le chemin vers le schéma W3C dans les documents est relatif au conteneur. Chaque conteneur est stocké sous la forme d'un fichier dans le répertoire utilisateur (commençant par le répertoire `C:\Documents and Settings` sous Windows).

Exercice 2

Quelques manipulations

Tout d'abord, il vous faut installer la base de données (*http://www.oracle.com/database/berkeley-db/index.html*).

Nous nous mettons à la place d'un administrateur d'un parc informatique. Chaque machine est identifiée et ses caractéristiques sont stockées dans un document XML (RAM, capacité disque, carte vidéo…). Vous devrez mettre en place un conteneur pour stocker des documents contenant ces informations. Ce stockage ne se fera qu'avec validation suivant un schéma W3C. Il vous sera demandé de trouver les machines ayant le minimum de mémoire et celles, à l'inverse, en ayant le plus, dans l'idée de réaliser un transfert par la suite.

Gestion de la base Berkeley DB par programmation

Cette base de données est utilisable en C++, Java, Perl, Python, PHP, et Tcl. Nous allons ici illustrer quelques manipulations usuelles en Java (Javadoc : *http://www.oracle.com/technology/documentation/berkeley-db/xml/java/index.html*).

L'ajout d'un document XML

Voici un exemple de code Java utilisant le conteneur `test.dbxml` et ajoutant un document d'alias test :

```
import com.sleepycat.dbxml.XmlContainer;
import com.sleepycat.dbxml.XmlManager;
import com.sleepycat.dbxml.XmlUpdateContext;

public class Test {
public static void main(String[] args) throws Throwable {
XmlManager manager = new XmlManager();
XmlContainer conteneur = manager.openContainer( "C:\\Documents and Settings\\alex
➥\\test.dbxml" );

 String unDocument = "<carnet><personne nom='dupont' prenom='arthur'/></carnet>";

 XmlUpdateContext ctx = manager.createUpdateContext();
 conteneur.putDocument(
  "test",
```

```
  unDocument,
  ctx,
  null );
conteneur.close();
manager.close();
}
}
```

Nous avons utilisé la classe `XmlManager` pour accéder au conteneur `test.dbxml`, dont nous avons spécifié la localisation par la méthode `openContainer`. L'objet obtenu, de type `XmlContainer`, contient alors toutes les primitives pour altérer le conteneur. La première primitive que nous avons utilisée est l'ajout d'un document par l'instruction `putDocument`. Nous retrouvons les mêmes arguments qu'en ligne de commandes. L'objet de type `XmlUpdateContext` n'est pas clairement précisé dans la documentation JavaDoc ; on peut supposer qu'il sert à conserver une trace des opérations sur un conteneur.

Pour vérifier que notre document a bien été ajouté, nous pouvons vérifier sa présence en ligne de commandes :

```
dbxml> query 'doc("test.dbxml/test")'
1 objects returned for eager expression 'doc("test.dbxml/test")'
dbxml> print
<carnet><personne nom="dupont" prenom="arthur"/></carnet>
dbxml>
```

Le document que nous avons ajouté était dans une chaîne de caractères. Il est cependant possible de fournir un document par des micro-événements. Chacun d'eux va servir à construire une partie du document, en fonction des informations qui seront à disposition à un moment donné (ajouter un élément, ajouter un attribut…), ce qui évite d'avoir à construire un document XML complet avant ajout. Comme nous le verrons dans la partie dédiée à la programmation, cela aura du sens lors de l'usage d'un mode de traitement SAX avec un parseur XML.

Voici un exemple :

```
XmlManager manager = new XmlManager();
XmlContainer conteneur = manager.openContainer( "C:\\Documents and Settings\\alex\\
➥test.dbxml" );
XmlUpdateContext ctx = manager.createUpdateContext();

XmlDocument doc = manager.createDocument();
 doc.setName("test2");
XmlEventWriter writer =
 conteneur.putDocumentAsEventWriter(doc, ctx);
writer.writeStartDocument(null, null, null);
writer.writeStartElement("carnet", null, null, 0, false);
writer.writeStartElement("personne", null, null, 2, false);
writer.writeAttribute("nom", null, null, "dupond", true);
writer.writeAttribute("prenom", null, null, "rene", true);
writer.writeEndElement("personne", null, null);
writer.writeEndElement( "carnet", null, null );
```

```
writer.writeEndDocument();
conteneur.close();
manager.close();
```

Nous passons à notre conteneur un document vierge, représenté par un objet de type `XmlDocument`. Cet objet est ensuite inséré par l'instruction `putDocumentAsEventWriter`. En l'état actuel, notre document, bien qu'ajouté, nécessite d'être complété grâce à l'objet retourné de type `XmlEventWriter`. Un ensemble de primitives (`writeStartElement`, `writeEndElement`…) vont ensuite nous servir à définir le contenu.

Nous pouvons vérifier le résultat de l'insertion par la console :

```
dbxml> query 'doc("test.dbxml/test2")'
1 objects returned for eager expression 'doc("test.dbxml/test2")'
dbxml> print
<carnet><personne nom="dupond" prenom="rene"/></carnet>
```

Le document XML peut, bien entendu, être fourni par un flux, en provenance du disque, par exemple. Il est aussi possible de lui associer des métadonnées, comme nous l'avons vu en mode console.

Par exemple :

```
XmlManager manager = new XmlManager();
XmlContainer conteneur = manager.openContainer( "C:\\Documents and Settings\\alex
➥\\test.dbxml" );
XmlUpdateContext ctx = manager.createUpdateContext();
XmlDocument doc = manager.createDocument();
doc.setName("test3");
XmlInputStream input = manager.createLocalFileInputStream( "c:/carnet3.xml" );
doc.setContentAsXmlInputStream( input );
doc.setMetaData( "http://dbxmlExamples/metadata", "auteur", new XmlValue
➥( "A.Brillant" ) );
conteneur.putDocument( doc, ctx );
conteneur.close();
manager.close();
```

Ce flux est créé par la fonction `createLocalFileInputStream` à laquelle nous associons un chemin. Tout s'effectue ensuite dans un objet de type `XmlDocument` représentant notre document.

La suppression d'un document est réalisée par l'instruction `deleteDocument` sur le conteneur, en précisant l'alias du document à supprimer.

Le requêtage de documents XML

L'exécution d'une requête XPath/XQuery a lieu au niveau de l'objet `XmlManager`. Une requête possède un contexte de requête contenant, par exemple, les espaces de noms utiles ou les variables souhaitées.

Voici un exemple :

```
XmlManagerConfig managerConfig = new XmlManagerConfig();
```

```
XmlManager manager = new XmlManager();
XmlContainer conteneur = manager.openContainer( "C:\\Documents and Settings\\alex
➥\\test.dbxml" );
conteneur.addAlias( "test" );

XmlUpdateContext ctx = manager.createUpdateContext();
XmlQueryContext context = manager.createQueryContext();

String myQuery = "collection( 'test' )/carnet";
XmlResults results = manager.query(myQuery, context);

XmlValue value = results.next();
while (value != null) {
 XmlDocument theDoc = value.asDocument();
 String docName = theDoc.getName();
 String docString = value.asString();
 System.out.println( "Nom : " + docName );
 System.out.println( "Contenu : " + docString );
 value = results.next();
}

conteneur.close();
//manager.close();
```

Nous avons ouvert un conteneur, comme précédemment, mais nous avons également créé un alias le représentant, par l'instruction `addAlias`. Il est ensuite utilisé dans notre requête XQuery. L'objet `XMLResults` stocke les résultats de la requête et l'instruction `next` sert à se déplacer de résultat en résultat. À chaque résultat, les instructions `getName` et `asString` nous donnent respectivement le nom du document lié et son contenu.

Voici la sortie obtenue :

```
Nom : carnet1
Contenu : <carnet>
<personne nom="dupont" prenom="jean"/>
<personne nom="dupond" prenom="alex"/>
<personne nom="dupons" prenom="test"/></carnet>
Nom : carnet2
Contenu : <carnet>
<personne nom="brillant" prenom="alex"/>
<personne nom="briand" prenom="aristide"/>
</carnet>
Nom : test
Contenu : <carnet><personne nom="dupont" prenom="arthur"/></carnet>
Nom : test2
Contenu : <carnet><personne nom="dupond" prenom="rene"/></carnet>
```

Pas de fermeture de la base

À noter que l'instruction de fermeture (`close`) sur l'objet `XmlManager` a déclenché une exception pour une raison indéterminée.

La mise à jour d'un document XML

La mise à jour d'un document XML peut être effectuée à partir d'un alias de document. Dans l'exemple ci-dessous, nous remplaçons l'intégralité d'un document par un autre :

```
XmlManagerConfig managerConfig = new XmlManagerConfig();
XmlManager manager = new XmlManager();
XmlContainer conteneur = manager.openContainer( "C:\\Documents and Settings\\alex
➥\\test.dbxml" );
XmlDocument document = conteneur.getDocument( "test2" );

document.setContent( "<carnet><personne nom='dupont' prenom='rene'/></carnet>" );
XmlUpdateContext uc = manager.createUpdateContext();
conteneur.updateDocument(document, uc);

conteneur.close();
//manager.close();
```

L'instruction `getDocument` sur le conteneur nous retourne un document de type `XmlDocument` représentant le document à mettre à jour. L'instruction `setContent` sert à en altérer le contenu. Il reste ensuite à effectuer la mise à jour par `updateDocument`.

Correction des exercices

Exercice 1

Pour l'installation de Xindice, vous trouverez à l'adresse *http://www.abrillant.com/test/xindice/xindice.zip* un package comprenant Tomcat et Xindice. Il vous suffira d'exécuter Tomcat à partir du répertoire `bin` (`tomcat5.exe`).

Pour exécuter des commandes Xindice, placez-vous dans le répertoire `webapps\xindice\WEB-INF` en utilisant une console (commande `cmd.exe` pour ouvrir la console et `cd` pour se déplacer dans un répertoire). Nous avons créé le batch suivant, `xindice.bat`, qui doit être suivi d'une commande Xindice :

```
java -Xms16m -Xmx128m -Djava.endorsed.dirs=.\lib -Dxindice.home=.
-Dxindice.db.home=db -Dxindice.configuration=system.xml -Dorg.apache.commons.logging.
og=org.apache.commons.logging.impl.SimpleLog -Dorg.apache.commons.logging.simplelog.
defaultlog=INFO -Dcmd.home=. -classpath classes;lib\commons-logging-1.0.3.jar;
lib\xalan-2.5.2.jar;lib\xerces-2.6.0.jar;lib\xml-apis.jar;lib\xmldb-api-20030701.jar;
lib\xmldb-api-sdk-20030701.jar;lib\xmldb-common-20030701.jar;
lib\xmldb-xupdate-20040205.jar;lib\xmlrpc-1.1.jar org.
apache.xindice.tools.XMLTools %*
```

Voici deux exemples de documents XML :

membre1.xml

```
<membre>
<nom>Dupont</nom>
<prenom>Jean</prenom>
<age>44</age>
```

```
<telephone>12345678</telephone>
<activites>
    <activite>Tennis</activite>
    <activite>Foot</activite>
</activites>
</membre>
```

membre2.xml

```
<membre>
<nom>Dupond</nom>
<prenom>Rene</prenom>
<age>44</age>
<telephone>12345678</telephone>
<activites>
    <activite>Vélo</activite>
    <activite>Foot</activite>
</activites>
</membre>
```

Nous commençons par ajouter une collection membres :

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice
➥\WEB-INF>xindice add_collection -c xmldb:xindice://localhost:8080/db -n membres
trying to register database
Created : xmldb:xindice://localhost:8080/db/members
```

Puis nous ajoutons les deux membres :

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice\WEB-INF>x
indice add_document -c xmldb:xindice://localhost:8080/db/membres -f c:\livreEyrolles\
➥membre1.xml -n membre1
trying to register database
Added document xmldb:xindice://localhost:8080/db/membres/membre1
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice\WEB-INF
➥xindice add_document -c xmldb:xindice://localhost:8080/db/membres -f
➥c:\livreEyrolles\membre2.xml -n membre2
trying to register database
Added document xmldb:xindice://localhost:8080/db/membres/membre2
```

Enfin, nous comptons les membres pour chaque activité :

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice\WEB-INF>x
➥indice xpath_query -c xmldb:xindice://localhost:8080/db/membres -q
➥"count(//activite[.='Foot'])"
trying to register database
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre1">1.0</xq:result>
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre2">0.0</xq:result>
```

Pour l'activité tennis :

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice\WEB-INF>x
indice xpath_query -c xmldb:xindice://localhost:8080/db/membres -q "count(//acti
vite[.='Tennis'])"
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre1">0.0</xq:result>
```

```
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre2">1.0</xq:result>
```

Pour l'activité vélo :

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\xindice\WEB-INF>x
indice xpath_query -c xmldb:xindice://localhost:8080/db/membres -q
➥"count(//activite[.='Vélo'])"
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre1">0.0</xq:result>
<xq:result xmlns:xq="http://xml.apache.org/xindice/Query" xq:col="/db/membres" x
q:key="membre2">1.0</xq:result>
```

À noter le caractère peu pratique des requêtes, qui ne permet pas d'effectuer une requête sur l'ensemble des documents de la collection.

Exercice 2

L'installation ne présente pas de difficultés particulières ; il est plus simple de passer par le fichier MSI sous Windows (installation avec mise à jour du menu Démarrer). Un menu Berkeley DB XML est alors ajouté et comporte la commande `Berkeley DB XML Shell` (`dbxml.exe`).

Commençons par créer le conteneur avec validation :

```
dbxml> createcontainer parc.dbxml d validate
Creating document storage container, with validation
```

Voici le schéma que nous avons réalisé pour valider le document :

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

<xs:element name="machine">

<xs:complexType>

 <xs:sequence>
  <xs:element name="ram" type="xs:int" maxOccurs="4"/>
  <xs:element name="disque" type="xs:int" maxOccurs="4"/>
   <xs:element name="carte" minOccurs="0" maxOccurs="5">
    <xs:complexType>
     <xs:simpleContent>
      <xs:extension base="xs:string">
       <xs:attribute name="type" type="xs:string"/>
      </xs:extension>
     </xs:simpleContent>
    </xs:complexType>
   </xs:element>
  </xs:sequence>

<xs:attribute name="id" type="xs:string"/>

</xs:complexType>

</xs:element>
</xs:schema>
```

Ce schéma est déposé au même niveau que le conteneur `parc.dbxml` (répertoire `C:/Documents and Settings/Votre compte Utilisateur/m.xsd`).

Voici deux documents, `m1.xml` et `m2.xml`, pour le conteneur :

```
M1.xml :
<machine id="m1" xsi:noNamespaceSchemaLocation="m.xsd" xmlns:xsi=
➥"http://www.w3.org/2001/XMLSchema-instance">
<ram>512</ram>
<ram>512</ram>
<disque>500</disque>
<disque>1000</disque>
<carte type="video">nvidia</carte>
<carte type="son">sound blaster</carte>
</machine>
M2.xml :
<machine id="m2" xsi:noNamespaceSchemaLocation="m.xsd" xmlns:xsi=
➥"http://www.w3.org/2001/XMLSchema-instance">
<ram>512</ram>
<disque>5000</disque>
<disque>1000</disque>
<carte type="video">nvidia</carte>
<carte type="son">sound blaster</carte>
</machine>
```

Les occurrences de RAM correspondent aux différentes barrettes mémoire. Ajoutons maintenant les documents dans le conteneur.

```
dbxml> putdocument m1 'c:/livreEyrolles/m1.xml' f
Document added, name = m1
dbxml> putdocument m2 'c:/livreEyrolles/m2.xml' f
Document added, name = m2
```

Effectuons alors une recherche des machines ayant le moins de mémoire, puis de celles ayant le plus de mémoire :

```
dbxml> query 'min( for $i in collection("parc.dbxml")/machine
return sum($i/ram))'
1 objects returned for eager expression 'min( for $i in collection("parc.dbxml")
/machine
return sum($i/ram))'
dbxml> print
512
dbxml> query 'max( for $i in collection("parc.dbxml")/machine
return sum($i/ram))'
1 objects returned for eager expression 'max( for $i in collection("parc.dbxml")
/machine
return sum($i/ram))'
dbxml> print
1024
```

8

Programmation XML

Ce chapitre expose des notions fondamentales pour programmer avec XML. Nous traiterons de SAX et de DOM, avec un tronc commun à la plupart des langages de programmation (PHP, ASP, JavaScript, Java…), puis nous aborderons des concepts plus particuliers, comme les techniques de mapping avec JAXB destinées à simplifier le développement et à rendre XML pratiquement transparent pour le développeur.

Son rôle

Programmer avec XML ne doit pas être considéré comme quelque chose de marginal. Aujourd'hui, tout développeur doit avoir dans sa panoplie de compétences un bagage minimal pour travailler avec XML. La programmation avec XML n'a rien de complexe en soi, puisqu'on s'appuiera sur une librairie (API) simplifiant une grande partie du travail. Cependant, quelle que soit la puissance de l'API, certains modes opératoires vont nécessiter des choix technologiques. Par exemple, des traitements volumineux risquent d'interdire une représentation complète du document en mémoire au profit d'un sous-ensemble choisi en fonction du traitement.

Un cas classique d'usage de XML est la constitution du fichier de configuration d'une application. Ce fichier de configuration donnera, dans un cadre d'exploitation, une certaine liberté fixée par le développeur (apparence graphique, localisation de la base de données…).

Autre cas de programmation : les imports/exports. Votre application est associée à des informations que l'on souhaite pouvoir injecter dans d'autres systèmes. Il faudra donc être capable de créer un document XML propre. À l'inverse, d'autres systèmes pourront eux-mêmes alimenter votre application XML qui devient alors un pont commun entre différents systèmes.

Les parseurs XML

Le parseur est l'élément de programmation le plus important, puisque c'est lui qui réalise le travail d'analyse du document XML. Son rôle est de vérifier la cohérence du document XML (en termes syntaxique et/ou par rapport à un schéma ou une DTD) et de transmettre à l'application les informations utiles au traitement du document. Tous les parseurs n'offrent pas la même puissance, certains étant non validants et d'autres validants. De plus, certains supportent une API SAX ou bien DOM, qui permet au parseur d'entrer en communication avec l'application, comme nous le verrons un peu plus tard.

Pour une application, un parseur fait partie d'une API ou d'une bibliothèque : ce n'est pas un organe indépendant mais un bloc de traitement que l'on peut contrôler, comme n'importe quelle méthode externe.

Vous trouverez dans le premier chapitre quelques références de parseurs XML.

La technologie SAX

SAX (*Simple API for XML*) définit un mode de communication entre le parseur et l'application, lié à un mécanisme événementiel. C'est un projet Open Source (*http://sax.sourceforge.net/*) qui propose deux versions d'une API, où seule la deuxième version est capable de prendre en compte les espaces de noms.

Plutôt que de représenter la totalité du document XML en mémoire, le parseur réalise un découpage du document en petites unités et transmet ces unités à l'application au fur et à mesure de l'analyse du document. Une unité représentera, par exemple, l'ouverture d'un élément ou sa fermeture, la rencontre d'un texte… On le comprendra, dans ce système, l'application n'a pas de représentation globale du document ou plutôt le parseur ne fournit qu'un cheminement dans le document, que l'application est libre de stocker ou non.

Les avantages et inconvénients de SAX

Les avantages de SAX sont bien sûr liés à la vitesse de traitement et à la faible consommation mémoire côté parseur. On le retiendra sur des traitements de documents de taille importante. La contrepartie est l'accroissement de la complexité de la programmation et l'impossibilité de pratiquer des requêtes par XPath/XQuery, qui nécessite une représentation globale en mémoire.

Voici une liste de quelques parseurs supportant SAX :

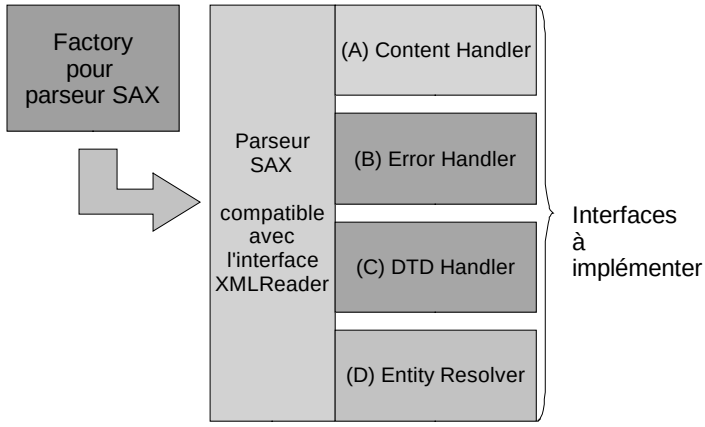
- Crimson 1.1.3 : présent dans le JDK 1.4. Support de SAX2 / DOM et JAXP.
- GNU JAXP 1.0b1 w/AElfred2 : SAX2 et JAXP 1.1.
- Oracle V2 (9.2.0.6) : SAX2 / DOM / JAXP.
- Piccolo 1.04 : SAX2 et JAXP.
- Resin 3.0.8 XML : SAX2 et JAXP.
- Xerces 2.6.2 : SAX2 / DOM / JAXP.
- XP 0.5 : SAX1.
- XPP3 1.1.3.4.G : Pull Parser / SAX wrapper.
- FastParser : SAX2 / DOM / JAXP.

Programmer avec SAX

Nous allons ici analyser les différentes composantes de SAX. À noter qu'il faut être un peu familiarisé avec le terme interface. Cette notion désigne simplement un ensemble de méthodes dont il faut respecter les règles d'utilisation et dont le contenu sera défini dans une classe d'implémentation.

Les différentes parties de SAX

Figure 8-1
Interfaces SAX



Dans la figure 8-1, nous avons représenté les quatre interfaces disponibles avec SAX. Si vous n'êtes pas familiarisé avec la notion d'interface, qui est propre à la programmation objet, retenez qu'il s'agit d'une sorte de classe abstraite (pas de code, juste des signatures de méthodes) qui assure la communication entre deux applications ou parties d'une même application. Chaque application s'appuie sur l'interface, ce qui garantit la cohérence

des appels de méthodes. Ainsi, une application qui connaîtra une interface n'aura pas besoin de connaître la nature d'une autre application utilisant cette même interface : c'est, en quelque sorte, une programmation par contrat.

L'interface principale : Content Handler

Content Handler est l'interface centrale, car c'est elle qui sert à la communication avec l'application. Son rôle est de transmettre les événements du parseur à l'application. Elle comporte autant de méthodes que de formes de représentation XML (ouverture et fermeture des éléments, espaces de noms, textes, commentaires…).

Voici un extrait de ces méthodes, sachant que nous avons choisi une représentation Java qui est facilement transposable dans divers langages.

Réception du texte :

```
Package org.xml.sax :
void characters(char[] ch, int start, int length)
```

Notification de fin de document :

```
void endDocument()
```

Notification de fin de balise :

```
void endElement(java.lang.String uri, java.lang.String localName, java.lang.String qName)
```

Notification de fin de portée d'un préfixe :

```
void endPrefixMapping(java.lang.String prefix)
```

Notification de blancs ignorés :

```
void ignorableWhitespace(char[] ch, int start, int length)
```

Notification d'une instruction de traitement :

```
void processingInstruction(java.lang.String target, java.lang.String data)
```

Objet utilisé pour la localisation du parsing (colonne, ligne) :

```
void setDocumentLocator(Locator locator)
```

Entité ignorée :

```
void skippedEntity(java.lang.String name)
```

Notification de début du document :

```
void startDocument()
```

Notification de début de balise :

```
void startElement(java.lang.String uri, java.lang.String localName,
java.lang.String qName, Attributes atts)
```

Rapporte la déclaration d'un préfixe pour un espace de noms :

```
public void startPrefixMapping(String prefix, String uri)
```

Nous allons maintenant observer, sur le document XML suivant, comment ces différentes méthodes sont invoquées :

```
<?xml version="1.0"?>
```

```
<carnet>
<personne id="p1">
 <nom>Fog</nom>
 <prenom>Phileas</prenom>
</personne>
<personne id="p2">
 <nom>Partout</nom>
 <prenom>Passe</prenom>
</personne>
</carnet>
```

En ayant suivi la trace de chaque méthode déclenchée, nous avons obtenu ceci :

```
Locator: org.apache.crimson.parser.Parser2$DocLocator@1004901
start document
startElement  carnet carnet org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters []
characters [
]
characters []
startElement  personne personne org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters []
characters [
]
characters []
startElement  nom nom org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters [Fog]
endElement :  nom nom
characters []
characters [
]
characters []
startElement  prenom prenom org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters [Phileas]
endElement :  prenom prenom
characters []
characters [
]
characters []
endElement :  personne
characters []
startElement  personne personne org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters []
characters [
]
characters []
startElement  nom nom org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters [Partout]
endElement :  nom nom
characters []
characters [
]
```

```
characters []
startElement  prenom prenom org.apache.crimson.parser.AttributesExImpl@18fe7c3
characters [Passe]
endElement :  prenom prenom
characters []
characters [
]
characters []
endElement :  personne personne
characters []
characters [
]
endElement :  carnet carnet
end document
```

Nous pouvons observer que la méthode `setDocumentLocator` est invoquée en premier. Cela semble logique, puisqu'elle fournit un objet donnant au fur et à mesure la localisation (ligne et colonne) dans le document. Les méthodes `startDocument` et `endDocument` bornent tous les autres appels. Ces méthodes pourraient donc servir à réaliser l'initialisation et la libération de ressources (connexion vers une base, ouverture de fichier…). Nous rencontrons ensuite une succession d'ouvertures et fermetures d'éléments par les méthodes `startElement` et `endElement`. Chaque bloc de texte déclenche la méthode `characters`. Cela inclut même les blancs (retour à la ligne, tabulation…) qui séparent les balises.

Voici un extrait de code Java vous montrant comment l'appel au parseur a été réalisé :

```
package sax;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public class Test1 {
public static void main( String[] args ) throws Exception {
 // Choix du parseur
  System.setProperty(
   "org.xml.sax.driver",
   "org.apache.crimson.parser.XMLReaderImpl" );

  XMLReader xr = XMLReaderFactory.createXMLReader();
  xr.setContentHandler( new Reader1() );
  xr.parse( "c:.../carnet.xml" );
 }
```

Le code commence par la méthode `main`, où on choisit un parseur compatible SAX (ici `org.apache.crimson.parser.XMLReaderImpl`). Ce parseur est compatible avec une interface `XMLReader`, cette dernière contenant entre autres la méthode `setContentHandler`, qui sera associée avec l'objet recevant les événements de parsing. L'objet en question est décrit ci-après ; il doit respecter l'interface `ContentHandler` avec autant de méthodes que d'événements de parsing (ouverture d'élément…).

```
package sax;

import org.xml.sax.Attributes;
```

```
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;

public class Reader1 implements ContentHandler {
public void endDocument() throws SAXException {
   System.out.println( "end document" );
}

public void startDocument() throws SAXException {
   System.out.println( "start document" );
}

public void characters(char[] ch, int start, int length)
     throws SAXException {
   System.out.println( "characters [" + new String( ch, start, length ) + "]" );
}
...
}
```

L'interface de localisation des événements : Locator

L'interface `Locator` donne des indications sur la position du parseur dans le document source. L'objet lié à cette interface (appelé implémentation ou réalisation) est fourni par la méthode `setDocumentLocator`, que nous avons vue précédemment.

Voici les principales méthodes :

- numéro de colonne :

```
int getColumnNumber()
```

- numéro de ligne :

```
int getLineNumber()
```

- Localisation publique (un identifiant) :

```
java.lang.String getPublicId()
```

- Localisation système (le path ou l'URI) :

```
java.lang.String getSystemId()
```

La représentation des attributs par l'interface Attributes

Cette interface correspond à une collection d'attributs. Un objet lié à cette interface est fourni par la méthode `startElement` (dernier argument), que nous avons vue précédemment.

Voici ces méthodes :

- retourne l'index d'un nom d'attribut qualifié (avec préfixe) :

```
int getIndex(java.lang.String qName)
```

- retourne l'index d'un nom d'attribut couplé à l'URI de son espace de noms :

```
int getIndex(java.lang.String uri, java.lang.String localName)
```

- retourne le nombre d'attributs :

```
int getLength()
```

- retourne le nom local (sans préfixe) selon un index :

```
java.lang.String getLocalName(int index)
```

- retourne le nom qualifié selon un index :

```
java.lang.String getQName(int index)
```

- retourne le type d'un attribut (CDATA, ID, IDREF...) selon un index :

```
java.lang.String getType(int index)
```

- retourne le type d'un attribut (CDATA, ID, IDREF...) selon un nom qualifié (préfixé) :

```
java.lang.String getType(java.lang.String qName)
```

- retourne le type d'un attribut (CDATA, ID, IDREF...) selon un espace nom :

```
java.lang.String getType(java.lang.String uri, java.lang.String localName)
```

- retourne l'URI lié à l'espace de noms de l'attribut selon l'index :

```
java.lang.String getURI(int index)
```

- retourne la valeur d'un attribut selon l'index :

```
java.lang.String getValue(int index)
```

- retourne la valeur d'un attribut selon son nom qualifié (`qName`) :

```
java.lang.String getValue(java.lang.String qName)
```

L'utilisation des features pour guider le parseur

Les *features* représentent un moyen de configurer le parseur. On peut les voir comme un ensemble de *flags* (drapeaux) à activer ou désactiver (donc de nature booléenne). On les positionne par la méthode `setFeature` sur un objet compatible avec l'interface `XMLReader` (le parseur SAX). En cas d'erreur, des exceptions `SAXNotRecognizedException` et `SAXNotSupportedException` peuvent être levées (une exception en Java est un moyen de signaler une erreur).

Voici quelques features :

- `http://xml.org/sax/features/namespaces` : gestion des espaces de noms ;
- `http://xml.org/sax/features/namespace-prefixes` : gestion des préfixes des espaces de noms ;
- `http://xml.org/sax/features/validation` : validation (DTD/Schéma...).

Vous trouverez d'autres features à l'adresse : *http://www.saxproject.org/apidoc/org/xml/sax/package-summary.html*. L'état d'une feature peut également être vérifié par l'instruction `getFeature`.

Le parseur Xerces dispose aussi de nombreuses features dont vous trouverez une liste exhaustive à l'adresse : *http://xerces.apache.org/xerces-j/features.html*. À titre d'exemple, nous trouvons :

- `http://apache.org/xml/features/validation/schema` : valide avec un schéma XML ;
- `http://apache.org/xml/features/validation/schema-full-checking` : vérifie l'intégralité du schéma (peut être coûteux en temps) ;
- `http://apache.org/xml/features/nonvalidating/load-external-dtd` : autorise le chargement d'une DTD externe.

L'utilisation des propriétés pour guider le parseur

Les propriétés sont proches des features. Il s'agit simplement de valeurs que l'on passe au parseur (interface `XMLReader`) par la méthode `setProperty`. Là aussi, une liste exhaustive est disponible sur le site officiel (*http://www.saxproject.org/apidoc/org/xml/sax/package-summary.html*). À titre d'exemple, le parseur Xerces propose les propriétés suivantes :

- `http://apache.org/xml/properties/schema/external-schemaLocation` : URI d'un schéma lié à l'espace de noms principal (celui de la racine) du document parsé. Ce schéma servira à valider le document.
- `http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation` : URI d'un schéma sans espace de noms. Ce schéma servira à valider le document.

Conséquence sur la gestion des blancs

Comme nous l'avons vu précédemment, tous les caractères blancs (même ceux séparant les balises) sont perçus comme faisant partie d'un texte, car rien n'indique au parseur s'il s'agit d'un contenu mixte ou non. Pour éviter cela, il suffit de valider le document XML par une DTD ou un schéma, qui définit précisément où doit se situer le texte et permettra au parseur d'ignorer les blancs séparant les différentes balises.

Voici comment procéder :

```
XMLReader xr = XMLReaderFactory.createXMLReader();
xr.setFeature( "http://xml.org/sax/features/validation", true );
```

Dans cet extrait de code, nous avons simplement activé la validation. En examinant la trace des notifications du parseur, nous obtenons :

```
Locator : org.apache.crimson.parser.Parser2$DocLocator@1004901
start document
startElement  livre livre org.apache.crimson.parser.AttributesExImpl@1a1c887
ignorableWhitespace []
ignorableWhitespace [
]
```

Comme vous pouvez le remarquer, les blancs séparant nos balises sont transcrits par `ignorableWhitespace`.

Exercice 1

Affichage d'un document XML avec SAX

L'objectif de cet exercice est d'afficher certaines parties d'un document XML dont la structure est la suivante :

```
<livre>
<auteurs>
 <auteur nom="" prenom=""/>
</auteurs>
<sections>
 <section titre="">
  <chapitre titre="">
  <paragraphe>…</paragraphe>
  </chapitre>
 </section>
</sections>
</livre>
```

Le document pourra contenir plusieurs auteurs, sections, chapitres ou paragraphes.

À l'issu de l'exécution du programme utilisant SAX, le plan du livre devra être affiché avec la liste des auteurs :

```
Livre:Mon livre
Auteur:nom1 prenom1
Auteur:nom2 prenom2
1. Section1
1.1. Chapitre1
2. Section2
2.1. Chapitre1
2.2. Chapitre2
```

La gestion des erreurs par l'interface ErrorHandler

Cette interface sert à récupérer les erreurs de parsing un peu à la manière d'une épuisette. Nous avons trois niveaux d'erreurs :

- Avertissement : ce n'est pas en soi une erreur, mais l'indication d'une incohérence, comme la présence de définitions inutilisées dans une DTD (élément indépendant…).
- Erreur : il s'agit d'une erreur de validation liée à un schéma ou une DTD, ce qui n'empêche pas le parseur de continuer l'analyse du document.
- Erreur fatale : il s'agit d'une erreur de syntaxe qui ne permet plus au parseur de continuer son travail.

Nous retrouvons ces trois erreurs dans les trois méthodes de cette interface :

```
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
void warning(SAXParseException exception)
```

Remarque sur la gestion des erreurs

Il est bien sûr possible de ne pas continuer le parsing en cas d'avertissement ou d'erreur en levant une exception `SAXException`.

L'objet de type `SAXParserException`, passé en argument des méthodes précédentes, contient des informations sur la nature et la localisation de l'erreur. Voici les méthodes à disposition :

- colonne de l'erreur :

```
int getColumnNumber()
```

- ligne de l'erreur :

```
int getLineNumber()
```

- identifiant public du document :

```
java.lang.String getPublicId()
```

- identifiant système du document (path ou URI) :

```
java.lang.String getSystemId()
```

- message d'erreur :

```
java.lang.String getMessage()
```

Voici maintenant un exemple qui vous montre comment cette gestion d'erreurs peut être développée :

```
class MesErreurs implements ErrorHandler {
public void error(SAXParseException exception) throws SAXException {
 System.out.println( "Erreur ligne " + exception.getLineNumber()
 ➥+ " : " + exception.getMessage() );
}
public void fatalError(SAXParseException exception) throws SAXException {
 System.out.println( "Erreur fatale ligne " + exception.getLineNumber()
 ➥+ " : " + exception.getMessage() );
}
public void warning(SAXParseException exception) throws SAXException {
 System.out.println( "Avertissement ligne " + exception.getLineNumber()
 ➥+ " : " + exception.getMessage() );
}
}
```

```
...
XMLReader xr = XMLReaderFactory.createXMLReader();
xr.setFeature( "http://xml.org/sax/features/validation", true );
xr.setErrorHandler( new MesErreurs() );
...
```

La première partie de cet extrait concerne la classe `MesErreurs` qui respecte l'interface `ErrorHandler`. Nous nous sommes contentés d'afficher chaque message d'erreur avec sa localisation dans le document source. La deuxième partie utilise la méthode `setErrorHandler` pour connecter le parseur à cette classe (par l'objet créé avec l'opérateur `new`) de traitement des erreurs.

Les interfaces DTDHandler et Entity Resolver

L'interface `DTDHandler` est particulière. Elle est liée à l'analyse de la DTD par le parseur.

L'interface `EntityResolver` comprend la méthode suivante :

```
InputSource resolveEntity(java.lang.String publicId, java.lang.String systemId)
```

Celle-ci sert à indiquer au parseur comment trouver certaines ressources (DTD…) ou solutions employées pour les catalogues. Ces derniers servent à travailler localement avec un document dont le schéma ou la DTD est normalement accessible sur Internet. Le parseur Xerces supporte cette gestion par catalogue (*http://www.oasis-open.org/committees/entity/spec-2001-08-06.html*).

Amélioration de la programmation par les filtres

L'idée sous-jacente à l'utilisation d'un filtre est de modifier, par programmation, les événements SAX sans changer le document XML analysé. L'application ne fait ainsi plus de distinction avec un document XML classique. Son caractère reste cependant marginal mais apporte un confort de programmation indéniable, puisque cela évite de changer le code de traitement lorsqu'un document XML n'est pas tout à fait structuré comme il le faudrait.

Tout fonctionne avec une classe `XMLFilterImpl` qui joue un peu le rôle de parseur tout en implémentant l'interface `ContentHandler`. En tant que développeur, il vous suffit de surcharger (réécrire) la méthode correspondant à l'événement SAX que vous voulez changer.

```
class Filtre extends XMLFilterImpl {
public Filtre( XMLReader reader ) {
    super( reader );
}
public void startElement(String uri,
    String localName,
    String qName,
    Attributes atts ) throws SAXException {
    if ( "personne".equals( localName ) )
        localName = "PERSONNE";
    super.startElement(uri, localName, qName, atts);
    } }
```

Ici, nous avons surchargé la méthode `startElement` dans l'optique que l'élément `personne` deviennent l'élément `PERSONNE` pour l'application.

Si maintenant nous reprenons notre programme de trace, sans en changer le contenu, nous allons obtenir ceci :

```
XMLReader xr = XMLReaderFactory.createXMLReader();
Filtre filter = new Filtre( xr );
filter.setContentHandler( new Reader1() );
filter.parse( "carnet.xml" );
```

```
startElement  PERSONNE personne org.apache.crimson.parser.AttributesExImpl@13e8d89
characters []
characters [
]
...
```

Exercice 2

Utilisation d'un filtre

Créez une implémentation de l'interface `ContentHandler` effectuant un affichage simple de tous les textes dans la console. Vous utiliserez, par exemple, le document XML de l'exercice précédent pour vos tests.

Créez maintenant un filtre effectuant une conversion de tous les textes en majuscules. Utilisez l'implémentation précédente pour vos tests.

La technologie Java : JAXP

JAXP (*Java API for XML Processing*) est une brique de la plate-forme Java. C'est un système qui masque le parseur véritablement employé, un peu dans l'esprit des pilotes que l'on retrouve dans les connexions aux bases de données. Un parseur Java qui souhaite devenir compatible JAXP doit présenter quelques classes supplémentaires et notamment hériter de la classe `javax.xml.parsers.SAXParser`.

Parsing et validation de documents XML

Voici quelques exemples d'utilisation :

Le premier cas est un parsing en mode SAX avec validation par rapport à une DTD.

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.XMLReader;

public class JAXPSAX {
public static void main(String[] args) throws Throwable {
 SAXParserFactory factory = SAXParserFactory.newInstance();
 factory.setValidating( true );
```

```
 SAXParser sp = factory.newSAXParser();
 XMLReader reader = sp.getXMLReader();
 reader.setContentHandler( ... );
 reader.parse( "livre.xml" );
}
}
```

La classe `SAXParserFactory` masque en réalité la façon dont le parseur va être construit. L'objet retourné, de type `SAXParserFactory`, dispose d'une méthode `setValidating` pour garantir la validation en cas d'usage d'une DTD. Toute la suite est très similaire à ce que nous avons vu précédemment, car la méthode `getXMLReader` rend le parseur compatible SAX.

Si on souhaite valider le document par rapport à un schéma, il faut utiliser la propriété `http://java.sun.com/xml/jaxp/properties/schemaLanguage` selon cet exemple :

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating( true );
factory.setNamespaceAware( true );
SAXParser sp = factory.newSAXParser();
sp.setProperty(
"http://java.sun.com/xml/jaxp/properties/schemaLanguage",
http://www.w3.org/2001/XMLSchema" );
XMLReader reader = sp.getXMLReader();
reader.setErrorHandler( new MesErreurs() );
reader.setContentHandler( new Reader1() );
reader.parse( "livre.xml" );
```

La relation entre le document XML et le schéma s'effectue toujours de la même manière par les attributs `noNamespaceSchemaLocation` ou `schemaLocation`.

L'intégration d'un nouveau parseur JAXP

L'utilisation d'un autre parseur compatible JAXP peut être réalisé par configuration. Il suffit, pour cela, de mettre à jour la propriété de l'application `javax.xml.parsers.SAXParserFactor` avec la classe JAXP du parseur.

Prenons l'exemple du parseur Piccolo (*http://piccolo.sourceforge.net/*). Ce dernier possède une classe compatible JAXP `com.bluecast.xml.JAXPSAXParserFactory`. Bien entendu, on prendra garde à disposer de la librairie du parseur dans le `classpath` (`piccolo.jar`).

Pour exécuter une application avec ce parseur, il suffit d'ajouter dans la ligne de commande de démarrage l'option `-D javax.xml.parsers.SAXParserFactor= com.bluecast.xml.JAXPSAXParserFactory`.

Cela peut également être effectué dans l'application selon cet exemple :

```
System.setProperty(
"javax.xml.parsers.SAXParserFactory",
"com.bluecast.xml.JAXPSAXParserFactory" );

SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser sp = factory.newSAXParser();
```

```
XMLReader reader = sp.getXMLReader();
reader.setErrorHandler( new MesErreurs() );
reader.setContentHandler( new Reader1() );
reader.parse( "livre.xml" );
```

L'instruction `setProperty` doit être appelée une fois dans l'application (généralement, cela sera dans les premières lignes).

Il est également possible de modifier le fichier de propriétés propre au JRE pour que toutes les applications soient concernées. Ce fichier est localisé dans le répertoire `lib/jaxp.properties`.

Programmation SAX avec PHP

En PHP, nous disposons également des fonctions `xml_set_element_handler` et `xml_set_character_data_handler` qui ont un rôle très similaire à SAX, puisque nous allons pouvoir, au fur et à mesure du parsing, invoquer certaines méthodes.

```
function start_element($parseur, $nom, $attrs) {
   print "<b>Demarrage élément :</b> $nom<br />";
   print "<b>Attributs:</b>";
   foreach ($attrs as $att => $val) {
   print "$att = $val<br />";
   }
   print "<br />";
}
function end_element($parseur, $nom) {
   print "<b>Fin élément:</b> $nom<br /><br />";
}
function characters($parseur, $texte) {
   print "<p><i>$texte</i></p>";
}
$parseur = xml_parser_create();
xml_set_element_handler($parseur, "start_element",
                                    "end_element");
xml_set_character_data_handler($parseur, "characters");

if ( $file_stream = fopen( "test.xml", "r" ) ) {
   while ($data = fread($file_stream, 4096)) {
       $flag = xml_parse($parseur, $data, feof($file_stream));
       if (!$flag) {
           $code = xml_get_error_code($parseur);
           $texte = xml_error_string($code);
           $ligne = xml_get_current_line_number($parseur);
            die( "Erreur de parsing à la ligne $ligne: $texte" );
       }

   }

} else {
die( "Erreur d'ouverture du fichier test.xml" );
}
```

Dans cet exemple, notre interface `ContentHandler` SAX devient les trois méthodes `start_element`, `end_element` et `characters`. Les arguments de ces méthodes sont imposés. Nous retrouvons le nom (qualifié) pour l'élément et ses attributs. Le parseur est créé par la méthode `xml_parser_create`. Les méthodes à invoquer pendant le parsing sont précisées par `xml_set_element_handler` et `xml_set_character_data_handler`. Pour le parsing en lui-même, on ouvre un flux sur le fichier : les données sont fournies au fur et à mesure par l'instruction `xml_parse`. Cette dernière retourne la valeur à 0 en cas d'erreur. À noter que la fonction `utf8_decode` peut être employée avec les données lues pour garantir un traitement correct des caractères latins.

Programmer avec DOM

DOM (*Document Object Model*) est une représentation objet d'un document XML. Chaque classe (ou type) d'objets provient des types de nœuds dans un document XML (élément, texte…). DOM est une API disponible dans la plupart des langages et maintenue par le W3C (*http://www.w3.org/DOM/*) ; la dernière version est le DOM Level 3.

DOM impose un modèle mémoire d'un document XML. Cela signifie qu'il est mal adapté à de gros documents XML. Dans tous les autres cas, cette représentation mémoire (par objet) offre beaucoup de souplesse en termes de navigation dans un document ou bien en termes de modification.

Pour la navigation, outre le parcours de fils en fils par l'API, ou bien sur les descendants, une requête XPath 1.0 ou XPath 2.0 (cette dernière version n'est pas présente sur toutes les plates-formes) est toujours réalisable.

Concernant les modifications, des changements de nœuds sont possibles et une opération de conversion des objets en document texte XML est disponible (sérialisation). De plus, les transformations XSLT peuvent également fonctionner sur cette arborescence. Pour conclure, DOM offre donc beaucoup de puissance. En revanche, la simplicité de l'API est parfois mise en cause de par son caractère multilangage, rognant sur les facilités de tel ou tel langage.

API DOM

Nous allons ici analyser les différentes parties de l'API DOM en commençant par la représentation des nœuds XML.

Les différents types de nœud

Voici les types d'objet disponibles (appelés interfaces par le W3C) :

- `Document` : cet objet représente l'ensemble du document. Il contient une référence (un objet de type `Element`) vers l'élément racine. Il sert également à construire différents types de nœud. Chaque nœud construit appartient à ce document par la propriété `ownerDocument`.

- `DocumentFragment` : un fragment de document est une sorte de mini-document. Son rôle consiste à stocker un ou plusieurs nœuds. Lorsqu'un fragment de document est ajouté dans un arbre, seuls ses enfants sont ajoutés. On peut imaginer son utilité dans des opérations de copier-coller entre divers arbres.
- `DocumentType` : cet objet est lié à une DTD et caractérise les entités internes ou externes du document (cela n'inclut pas les entités paramétriques).
- `EntityReference` : une entité référence est positionnée dans un texte et est liée à un objet `Entity`.
- `Element` : l'élément est probablement le nœud le plus employé et caractérise la balise. Il contient des primitives pour obtenir les nœuds fils et le parent ; il donne également accès aux attributs, soit directement, soit par l'intermédiaire de nœuds de type `Attr`.
- `Attr` : il s'agit d'un attribut d'élément avec un nom et une valeur.
- `ProcessingInstruction` : il s'agit d'une instruction de traitement. Il n'y a pas d'analyse ou de manipulation de l'instruction, seule la forme brute est disponible.
- `Comment` : il s'agit d'un commentaire ; généralement, ces nœuds sont éliminés par le parseur, mais tout dépend des propriétés de parsing.
- `Text` : c'est un nœud texte. À noter que si des nœuds texte sont adjacents, une procédure de normalisation les remplaçant par un seul nœud texte peut être utilisée en employant l'instruction `normalize` sur un nœud ancêtre.
- `CDATASection` : il s'agit d'une forme de nœud texte dont le contenu n'est pas analysé par le parseur (donc pas de conflit avec les caractères spéciaux < ou >, etc.). La normalisation que l'on retrouve dans les nœuds texte ne fonctionne pas nécessairement avec ce type de nœud.
- `Entity` : il s'agit d'une entité issue d'une DTD. Il est à noter que les parseurs ne supportant pas la validation peuvent ignorer ce type de nœud.
- `Notation` : il s'agit d'une notation d'une DTD. Une notation décrit le format d'une entité qui ne peut pas être analysée.

Chaque type de nœud peut avoir de zéro à n fils. Nous en faisons ici la synthèse :

```
Type de Nœud : Types de Nœuds fils
Document : Element (1 au plus), ProcessingInstruction, Comment, DocumentType (1 au plus)
DocumentFragment : Element, ProcessingInstruction, Comment, Text, CDATASection,
➥EntityReference
DocumentType : Pas d'enfant
EntityReference : Element, ProcessingInstruction, Comment, Text, CDATASection, Entity-
➥Reference
Element : Element, Text, Comment, ProcessingInstruction, CDATASection, Entity-
➥Reference
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
```

L'interface Node

Tous les nœuds ont des primitives communes par héritage de l'interface `Node`. Il s'agit d'un type générique qui, théoriquement, sert à ignorer la véritable nature d'un nœud. On peut imaginer que, sur une opération de copier-coller, la nature du nœud importe peu. On pourra alors privilégier cette interface.

Pour distinguer néanmoins la nature du nœud manipulé (c'est un peu redondant avec les capacités de certains langages en pensant, par exemple, à l'instruction `instanceof` en java, mais ne l'oublions pas DOM est multilangage), on dispose de la méthode `getNodeType` qui retourne une valeur entière associée aux constantes suivantes :

```
Node.ATTRIBUTE_NODE
Node.CDATA_SECTION_NODE
Node.COMMENT_NODE
Node.DOCUMENT_FRAGMENT_NODE
Node.DOCUMENT_NODE
Node.DOCUMENT_TYPE_NODE
Node.ELEMENT_NODE
Node.ENTITY_NODE
Node.ENTITY_REFERENCE_NODE
Node.NOTATION_NODE
Node.PROCESSING_INSTRUCTION_NODE
Node.TEXT_NODE
```

Trois méthodes courantes vont donner un résultat différent en fonction du nœud :

- `getNodeName` : le nom du nœud. S'il s'agit d'un élément, cela va correspondre à la balise ; dans le cas d'un attribut, au nom de l'attribut ; dans le cas d'une entité ou référence d'entité, au nom de l'entité ; et enfin, dans le cas d'une instruction de traitement, à son nom. Dans tous les autres cas, cette méthode n'aura pas vraiment d'usage.
- `getNodeValue` : si le nœud est un attribut, cela correspondra à la valeur de l'attribut. Pour toutes les formes de nœuds texte (texte simple, section CDATA), cela correspondra au texte lié. Il reste les commentaires et les instructions de traitement pour les autres valeurs. Dans tous les autres cas, cette méthode n'aura pas d'usage.
- `getAttributes` : seul l'élément est concerné. L'ensemble des nœuds attribut sera disponible via un objet de type `NamedNodeMap`. Pour tous les autres types de nœud, cette méthode n'aura pas d'effet.

Les opérations de lecture de l'arbre DOM

Voici quelques opérations en lecture d'usage courant sur l'interface `Node` :

```
NamedNodeMap getAttributes()
```

Liste des attributs (pour l'élément).

```
NodeList getChildNodes()
```

Liste des nœuds fils.

```
Node getFirstChild()
```

Le premier nœud fils.

```
Node getLastChild()
```

Le dernier nœud fils.

```
String getNamespaceURI()
```

L'URI, si un espace de noms est utilisé.

```
Node getNextSibling()/getPreviousSibling
```

Le nœud adjacent (avant ou après).

```
Document getOwnerDocument()
```

Le propriétaire du nœud (le document employé pour construire ce nœud).

```
Node getParentNode()
```

Le nœud parent.

```
String getPrefix()
```

Le préfixe.

```
String getTextContent()
```

Concaténation des textes contenus dans le nœud.

```
Object getUserData(String key)
```

Objet associé à la clé et au nœud.

```
String lookupNamespaceURI(String prefix)
```

URI associé au préfixe.

Les opérations de modification de l'arbre DOM

Voici maintenant quelques opérations modifiant l'arbre en mémoire via l'interface `Node` :

```
Node appendChild(Node newChild)
```

Ajouter un fils.

```
Node cloneNode(boolean deep)
```

Retourner un clone (complet ou non) du nœud.

```
Node insertBefore(Node newChild, Node refChild)
```

Insertion d'un nouveau nœud avant le fils `refChild`.

```
void normalize()
```

Élimination des textes adjacents pour avoir une structure homogène (un seul nœud texte).

```
Node removeChild(Node oldChild)
```

Suppression d'un nœud fils.

```
Node replaceChild(Node newChild, Node oldChild)
```

Remplacement d'un nœud fils par un nouveau.

```
void setNodeValue(String nodeValue)
```

Modification de la valeur du nœud ; l'opération va dépendre du type du nœud.

```
void setPrefix(String prefix)
```

Définir un préfixe pour ce nœud.

```
Object setUserData(String key, Object data, UserDataHandler handler)
```

Sert à affecter un objet quelconque au nœud en fonction d'une clé.

L'objet `handler` sert à recevoir des événements selon certaines modifications du nœud :

- clonage : utilisation de `cloneNode()` ;
- importation : utilisation de `Document.importNode()`, opération de clonage indirect ;
- suppression ;
- renommage : utilisation de `Document.renameNode()` ;
- adoption : `Document.adoptNode()`.

Remarque sur les opérations en écriture

Un nœud ne peut être ajouté à un document que s'il a pour propriétaire ce document. Dans le cas contraire, il est nécessaire d'utiliser la méthode `adoptNode` du document receveur.

L'interface racine : Document

Comme nous l'avons vu, cette interface caractérise l'ensemble du document. Voici quelques méthodes d'usage courant :

```
Element getDocumentElement()
```

Retourne l'élément racine (c'est-à-dire le premier élément du document).

```
Element getElementById(String elementId)
```

Retourne l'élément dont un attribut de type ID contient la valeur `elementId`.

```
NodeList getElementsByTagName(String tagname)
```

Retourne la liste des éléments ayant pour nom celui passé en argument. Cette méthode fonctionne avec tous les éléments de l'arbre.

```
adoptNode(Node node)
```

Changement de propriétaire du nœud passé en argument.

```
Element createElement(String tagName)
```

Créer un nouvel élément.

```
Text createTextNode(String data)
```

Créer un nouveau nœud texte.

```
createAttribute(String name)
```

Créer un nouveau nœud attribut.

```
createCDATASection(String data)
```

Créer un nouveau nœud texte (une section CDATA).

```
createEntityReference(String name)
```

Créer un nœud référence d'entité.

```
createComment(String data)
```

Créer un nouveau nœud commentaire.

```
createProcessingInstruction(String target,String data)
```

Création d'une instruction de traitement.

L'interface Element

Voici quelques méthodes d'usage courant :

```
String getAttribute(String name) / setAttribute( String name )
```

Accès / écriture d'un attribut.

```
NodeList getElementsByTagName(String name)
```

Retourne tous les éléments descendants du nom passé en argument.

```
void removeAttribute(String name)
```

Suppression d'un attribut.

```
String getTagName()
```

Nom de l'élément.

Pour la gestion des espaces de noms, il suffit d'ajouter `NS` au nom de la méthode puis de préciser l'espace de noms. Par exemple, la suppression d'un attribut dans un espace de noms correspond à la méthode `removeAttributeNS(String namespaceURI, String localName)`.

L'interface pour l'attribut : Attr

Un nœud attribut est associé à un élément par les méthodes suivantes :

```
Attr setAttributeNode(Attr newAttr)
Attr getAttributeNode(String name)
Attr removeAttributeNode(Attr oldAttr)
```

Voici quelques méthodes de l'interface `Attr` :

```
String getName()
```

Nom de l'attribut.

```
Element getOwnerElement()
```

Elément attaché.

```
boolean getSpecified()
```

Indique si l'attribut est lié à une valeur par défaut (DTD, schéma) ou non. Cela revient à savoir si l'attribut en question était bien dans le document XML analysé.

```
String getValue()
```

Valeur de l'attribut.

```
boolean isId()
```

Renvoie `true` si l'attribut est de type ID.

```
void setValue(String value)
```

Changement de la valeur de l'attribut.

L'interface Text

Cette interface hérite de l'interface `CharacterData`. Voici quelques méthodes courantes :

```
void appendData(String arg)
```

Ajouter un bloc de texte.

```
void deleteData(int offset, int count)
```

Suppression d'un texte à une position et une longueur données.

```
String getData() / setDate( String arg )
```

Le texte associé.

```
int getLength()
```

La longueur du texte.

```
void insertData(int offset, String arg)
```

Insertion du texte.

```
void replaceData(int offset, int count, String arg)
```

Remplacement d'une portion de texte par une autre à une position et une longueur données.

Remarque sur l'encodage

Le jeu de caractères que l'on obtient est généralement compatible UTF-16, le parseur réalisant la traduction avec le jeu de caractères du document source. Cela garantit un traitement universel du texte quel que soit le jeu de caractères de départ.

La technologie JAXP et DOM

Nous allons maintenant présenter comment, avec Java et sa solution JAXP, nous pouvons obtenir un document DOM.

Nous utiliserons le document XML, `carnet.xml`, suivant :

```
<?xml version="1.0"?>
<carnet>
 <personne id="p1">
  <nom>Fog</nom>
  <prenom>Phileas</prenom>
 </personne>
 <personne id="p2">
  <nom>Partout</nom>
  <prenom>Passe</prenom>
 </personne>
</carnet>
```

La création d'un parseur pour DOM

Avec JAXP, pour obtenir une référence sur l'élément racine `carnet`, nous écrivons le code suivant :

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

public class Test2 {
public static void main(String[] args) throws Throwable {
 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
 DocumentBuilder db = dbf.newDocumentBuilder();
// Partie DOM
 Document d = db.parse( "carnet.xml" );
 Element root = d.getDocumentElement();
 ...
}
}
```

L'objet `dbf` de cet exemple sert à construire l'accès au parseur `db` capable de produire un document DOM, via la méthode `parse`. JAXP garantissant une séparation entre le code d'usage et le code du parseur, il est possible de changer de parseur en toute transparence avec la propriété système `javax.xml.parsers.DocumentBuilderFactory` ou bien en modifiant le fichier du JRE `lib/jaxp.properties`.

La classe DocumentBuilderFactory

La classe `DocumentBuilderFactory` possède quelques flags pour affiner la construction de l'arbre DOM :

```
void setIgnoringComments(boolean ignoreComments)
```

Ignorer/gérer les commentaires.

```
void setIgnoringElementContentWhitespace(boolean whitespace)
```

Ignorer/gérer les blancs.

```
void setNamespaceAware(boolean awareness)
```

Ignorer/gérer les espaces de noms.

```
void setValidating(boolean validating)
```

Prendre en compte la DTD ou le schéma.

```
void setXIncludeAware(boolean state)
```

Prendre en compte `Xinclude` (système pour inclure un document dans un autre).

```
void setCoalescing(boolean coalescing)
```

Convertit les sections CDATA en texte et les concatène.

Il est également possible de lire/écrire des propriétés via les méthodes :

- `setFeature/getFeature` pour un booléen ;
- `setAttribute/getAttribute` pour les propriétés.

Le parseur Xerces propose, par exemple, ces propriétés :

```
http://apache.org/xml/features/dom/defer-node-expansion
```

Les nœuds ne sont dépliés que lors du parcours, ce qui peut améliorer les performances.

```
http://apache.org/xml/features/dom/create-entity-ref-nodes
```

Évite la création de nœud référence d'entité. Seul le texte associé à l'entité est présent.

```
http://apache.org/xml/features/dom/include-ignorable-whitespace
```

Ignorer ou non les blancs.

Le parcours d'un arbre DOM

Retournons maintenant à notre premier exemple Java et regardons comment nous pouvons parcourir les nœuds de notre document XML :

```
static void parcours( Element e ) {
 System.out.println( "-" + e.getNodeName() );
 // Parcours de tous les attributs
 NamedNodeMap nnm = e.getAttributes();
 for ( int i = 0; i < nnm.getLength(); i++ ) {
  Node n = nnm.item( i );
  System.out.println( "*" + n.getNodeName() +
  "=" + n.getNodeValue() );
 }
 // Parcours des fils
 NodeList nl = e.getChildNodes();
 for ( int i = 0; i < nl.getLength(); i++ ) {
  Node n = nl.item( i );
  if ( n.getNodeType() == Node.ELEMENT_NODE ) {
   parcours( ( Element )n );
  } else
   if ( n.getNodeType() == Node.TEXT_NODE ) {
    System.out.println( n.getNodeValue() );
   }
 }
}
```

La première partie de notre méthode parcourt tous les attributs de l'élément passé en argument grâce à l'objet de type `NamedNodeMap`. On affiche alors le nom et la valeur de chaque attribut. La deuxième partie permet de parcourir tous les nœuds fils grâce à un objet de type `NodeList`. Si l'élément fils est un élément, la méthode est appelée récursivement sur ce nœud ; s'il s'agit de texte, cela est alors affiché dans la console.

Voici la sortie console obtenue :

```
-carnet
-personne
*id=p1
-nom
Fog
-prenom
Phileas
-personne
*id=p2
-nom
Partout
-prenom
Passe
```

Exercice 3

Affichage d'un document XML

Le document XML est le même que dans l'exercice 1. L'objectif est aussi d'afficher le plan du livre, mais en ne vous appuyant que sur JAXP et DOM.

Les dangers du casting en lecture DOM

Certains modes de parcours de l'arbre peuvent se révéler dangereux et rendre votre code instable en cas de modification du document XML source.

Prenons cet exemple :

```
static void parcours( Element e ) {
 NodeList nl = e.getElementsByTagName( "personne" );
 for ( int i = 0; i < nl.getLength(); i++ ) {
  Element pers = ( Element )nl.item( i );
  Element nom = ( Element )pers.getFirstChild();
  System.out.println( nom.getNodeValue() );
 }
}
```

Ici le parcours revient, pour chaque élément `personne`, à prendre le premier nœud fils qui correspond à l'élément `nom`. Or, si le document XML n'a pas de schéma ou de DTD (c'est-à-dire sans règles pour indiquer au parseur si un blanc est un séparateur ou fait partie d'un texte), les blancs peuvent être pris en compte par le parseur et le premier nœud fils de l'élément `personne` devient alors un nœud texte. Cela aura pour conséquence de générer des erreurs dans le code.

Pour améliorer la qualité du code, il faut, lors des parcours, prendre en compte toutes les évolutions du document. Le plus simple pour éviter trop de permissivité est l'usage systématique d'une DTD ou d'un schéma.

La sérialisation d'un arbre DOM

La sérialisation est l'étape qui passe de notre représentation objet à une représentation physique sous forme de fichier. Curieusement, cette fonctionnalité pourtant banale n'a pas fait partie de l'API DOM jusqu'à l'arrivée de la version 3.

JAXP n'offre, pour sa part, pas de solution et il reste au codeur la possibilité artisanale de coder lui-même cette génération en parcourant l'arbre. Cependant, cette génération, dans l'ensemble peu complexe, se doit de traduire certaines parties du texte en entités (par exemple `<`…).

Reprenons notre exemple, où nous avons réalisé le code générique suivant :

```
static void parcours( Node n, PrintWriter pw ) {
 switch( n.getNodeType() ) {
  case Node.ELEMENT_NODE :
```

```
    pw.write( "<" + n.getNodeName() + ">" );
    NodeList all = n.getChildNodes();
    for ( int i = 0; i < all.getLength(); i++ ) {
     parcours( all.item( i ), pw );
    }
    pw.write( "</" + n.getNodeName() + ">" );
    break;
   case Node.TEXT_NODE :
    pw.write( n.getNodeValue() );
}
pw.flush();
}
```

Cet exemple revient à parcourir récursivement les éléments de l'arbre et à écrire, à chaque passage, dans un objet de type `PrintWriter` associé à un fichier. Si un élément est rencontré, la partie ouvrante, le contenu et la partie fermante sont enregistrés ; s'il s'agit simplement de texte, sa valeur est enregistrée (pour simplifier, ce programme ne traite pas les entités).

Nous avons alors obtenu le document suivant :

```
<carnet>
<personne>
 <nom>Fog</nom>
 <prenom>Phileas</prenom>
</personne>
<personne>
 <nom>Partout</nom>
 <prenom>Passe</prenom>
</personne>
</carnet>
```

Bien entendu, il faudrait améliorer notre code en ajoutant le prologue.

Il existe une classe `DOMSerializerImpl` dans la librairie Java qui peut réaliser ce travail. Malheureusement cette classe est issue du package `com.sun.org.apache.xml.internal.serialize` qui est à usage interne. Et il n'est pas garanti qu'elle sera présente dans les prochaines versions de Java.

Voici un exemple :

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setIgnoringElementContentWhitespace( true );
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse( "carnet.xml" );
DOMSerializerImpl serializer = new DOMSerializerImpl();
System.out.println( serializer.writeToString( d ) );
```

Comme vous pouvez le constater, c'est la méthode `writeToString` qui exécute le travail de conversion. Nous obtenons alors ce document :

```
<?xml version="1.0" encoding="UTF-16"?>
<carnet>
```

```
<personne id="p1">
 <nom>Fog</nom>
 <prenom>Phileas</prenom>
</personne>
<personne id="p2">
 <nom>Partout</nom>
 <prenom>Passe</prenom>
</personne>
</carnet>
```

Le parseur Xerces passe par des extensions DOM (de la version 3) pour offrir cette sérialisation sous la forme d'un service :

```
System.setProperty(DOMImplementationRegistry.PROPERTY,
"org.apache.xerces.dom.DOMImplementationSourceImpl");
DOMImplementationRegistry registry = DOMImplementationRegistry.newInstance();
DOMImplementationLS impl =
    ( DOMImplementationLS )registry.getDOMImplementation( "LS" );
LSSerializer ls = impl.createLSSerializer();
System.out.println( ls.writeToString( d ) );
```

La construction d'un nouveau document

Comme nous l'avons vu, nous disposons d'un ensemble de primitives via un objet de type `Document` pour construire l'arbre XML. Regardons, à l'aide de l'exemple ci-après, la manière de procéder :

```
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.newDocument();
Element root = d.createElement( "carnet" );
Element personne = d.createElement( "personne" );
personne.setAttribute( "id", "p1" );
Element nom = d.createElement( "nom" );
Text texte = d.createTextNode( "Fogg" );
nom.appendChild( texte );
Element prenom = d.createElement( "prenom" );
texte = d.createTextNode( "Phileas" );
prenom.appendChild( texte );
personne.appendChild( nom );
personne.appendChild( prenom );
root.appendChild( personne );
d.appendChild( root );
// Normalisation des nœuds textes
d.normalize();
```

Ici, nous obtenons un document vierge par l'instruction `newDocument`. Nous utilisons ensuite un ensemble de primitives, `createElement`, `createTextNode` et `appendChild`, pour reconstruire notre arborescence. La dernière méthode, `normalize`, évite la présence de nœuds texte adjacents.

Le document XML qui a été construit est le suivant :

```
<carnet>
<personne id="p1">
 <nom>Fogg</nom>
 <prenom>Phileas</prenom>
</personne>
</carnet>
```

Exercice 4

Toujours dans l'optique d'utiliser le format XML de l'exercice 1, créez une classe `Livre`, qui contiendra les primitives suivantes :

```
ajouterParagraphe( String titreChapitre, String texte)
```

Ajouter un paragraphe au chapitre.

```
ajouterChapitre( String titreSection, String titreChapitre )
```

Ajouter un chapitre à la section.

```
ajouterSection( String titreSection )
```

Ajouter une section.

```
afficherLivre()
```

Afficher dans la console le document XML final.

Testez ces méthodes avec un exemple et vérifiez le document XML produit dans la console par la méthode `afficherLivre`.

La fusion d'arbres DOM

Comme nous l'avons exposé, le passage de nœud d'un arbre à un autre nécessite un changement de propriétaire. La méthode `importNode` de l'interface `Document` réalise ce travail, mais effectue également une copie du nœud, ce qui permet de garder un document XML source, en copie seulement, sans provoquer d'altération.

Voici un exemple avec deux carnets XML. Nous recopions les éléments du carnet dans un autre :

```
Document carnet1 = db.parse( "carnet.xml" );
Document carnet2 = db.parse( "carnet2.xml" );
// Fusion des deux carnets
NodeList nlp = carnet1.getElementsByTagName(
"personne" );
for ( int i = 0; i < nlp.getLength(); i++ ) {
 Node personne1 = nlp.item( i );
 Node personne2 = carnet2.importNode( personne1, true );
 carnet2.getFirstChild().appendChild( personne2 );
}
```

Comme nous pouvons le noter, nous récupérons les éléments `personne` d'un premier carnet que nous insérons par duplication dans un deuxième document. À noter que l'argument `true` de la méthode `importNode` garantit que les nœuds descendants seront également dupliqués.

Exercice 5

On souhaite réaliser un document pour une bibliothèque.

Créez une classe pour agglomérer tous les livres d'un répertoire dans un nouveau document DOM de racine `biblio`. Vous afficherez le résultat dans la console.

```
<biblio>
<livre titre= "...">...</livre>
<livre tire="...">...</livre>
</biblio>
```

Si vous n'êtes pas familiarisé avec l'API Java, voici un exemple pour parcourir le contenu d'un répertoire :

```
File dir = new File( "path répertoire" );
String[] files = dir.list();
```

Programmation DOM avec PHP

Nous vous présentons ci-dessous un exemple de parcours d'une arborescence DOM en PHP. Nous sommes partis de l'exemple XML précédent, que nous avons affiché sous la forme d'une table HTML :

```
<html>
<body>
<?
$doc = new DOMDocument();
$doc->load( "carnet.xml" );
echo "<table border='1'>";
$personnes = $doc->getElementsByTagName( 'personne' );
for ($i = 0; $i < $personnes->length; $i++) {
    $enfants = $personnes->item( $i )->childNodes;
    $nom = $personnes->item( $i )->getElementsByTagName( 'nom' )->item( 0 )->nodeValue;
    // Ou
    // $nom = $enfants->item(0)->firstChild->nodeValue;
    $prenom = $personnes->item( $i )->getElementsByTagName( 'prenom' )->item( 0 )->nodeValue;
    // Ou
    // $prenom = $enfants->item(1)->firstChild->nodeValue;
    $id = $personnes->item($i)->getAttribute( "id" );
    echo "<tr><td>$id<td>$nom</td><td>$prenom</td></tr>";
}
echo "</table>"
```

```
?>
</body>
</html>
```

Ce code a produit le code HTML suivant :

```
<html>
<body>
<table border='1'><tr><td>p1</td><td>Fog</td><td>Phileas</td></tr><tr><td>p2</td><td>
➥Partout</td><td>Passe</td></tr></table></body>
</html>
```

La construction d'un nouvel arbre DOM

Dans cet exemple, nous avons construit le document DOM et avons opéré une sérialisation vers le fichier `test2.xml` :

```
<html>
<body>
<?
$doc = new DOMDocument();
$carnet = $doc->createElement( "carnet" );
$personne = $doc->createElement( "personne" );
$personne->setAttribute( "id", "1" );
$personne->setAttribute( "nom", "fogg" );
$personne->setAttribute( "prenom", "phileas" );
$carnet->appendChild( $personne );
$doc->appendChild( $carnet );
$doc->save( "test2.xml" );
?>
</body>
</html>
```

Le fichier `test2.xml` contient alors :

```
<?xml version="1.0"?>
<carnet><personne id="1" nom="fogg" prenom="phileas"/></carnet>
```

Nous aurions pu changer l'encodage, avant sauvegarde, par la propriété de même nom avec, par exemple, ce jeu de caractères : `$doc->encoding = "ISO-8859-1"`.

Voici quelques propriétés du document :

- `xmlVersion` : version XML (1.0) ;
- `doctype` : la déclaration du type de document ;
- `documentElement` : l'élément racine (propriété en lecture seulement) ;
- `preserveWhiteSpace` : supprimer ou non les espaces redondants ;
- `resolveExternals` : remplacer toutes les références à des entités externes par leurs valeurs.

Programmation DOM avec ASP

Voici le même exemple, mais réalisé avec ASP et s'appuyant indirectement sur MSXML :

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="1252"%>
<html>
<body>
<%
Dim doc
Set doc = Server.CreateObject("Microsoft.XMLDOM")
doc.async=false
doc.load(Server.MapPath("carnet.xml"))
if doc.parseError.errorcode<>0 then
 response.write( "Erreur de parsing " & doc.parseError.reason )
else
Response.write( "<table border='1'>" )
Set lst = doc.getElementsByTagName( "personne" )
For i = 0 to ( lst.length - 1 )
 nom = lst.item( i ).getElementsByTagName( "nom" ).item(0).text
 prenom = lst.item( i ).getElementsByTagName( "prenom" ).item(0).text
 Response.write( "<tr><td>" & nom & "</td><td>" & prenom & "</td></tr>" )
Next
Response.write( "</table>" )
end if
%>
</body>
</html>
```

Il faut noter que le path vers le fichier XML doit être donné en absolu, d'où l'appel à la fonction `Server.MapPath` pour effectuer la conversion. Autre particularité : la propriété `async` doit être initialisée à `false` pour garantir un parsing synchrone.

Nous avons obtenu la page HTML suivante :

```
<html>
<body>
<table border='1'><tr><td>Fog</td><td>Phileas</td></tr><tr><td>
➥Partout</td><td>Passe</td></tr></table>
</body>
</html>
```

Programmation DOM avec JavaScript

Bien que moins employé, JavaScript peut aussi être utilisé pour lire un document XML en s'appuyant sur l'API DOM. Cette technique est également présente dans l'architecture Ajax (*Asynchronous JavaScript and XML*). Il existe malheureusement des distinctions entre les navigateurs. Prenons l'exemple d'une lecture et l'affichage d'un document XML avec Internet Explorer :

```
<html>
<script type="text/javascript">
function afficherCarnet() {
 var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
 xmlDoc.load("carnet.xml");
 var doc = xmlDoc.documentElement;
 var noeuds = doc.childNodes;
 for ( i = 0; i < noeuds.length; i++ ) {
  if ( noeuds[ i ].nodeType == 1 ) { // Element
   document.write( "[" + noeuds[ i ].getAttribute( "nom" ) +
"]<br>" );
 }
}
}
</script>
<body>
<script type="text/javascript">
 afficherCarnet();
</script>
</body>
</html>
```

Dans cet exemple, les éléments de la racine (`documentElement`) ont été parcourus. Comme JavaScript ne dispose pas de solution pour vérifier la nature d'un objet, nous utilisons la propriété `nodeType` pour détecter la présence des éléments. Chaque nom est affiché à l'issu du parcours.

Sous Firefox, nous avons écrit :

```
<html>
<script type="text/javascript">
var xmlDoc = document.implementation.createDocument("", "", null);
function init() {
 xmlDoc.onload = afficherCarnet;
 xmlDoc.load("carnet.xml");
}
function afficherCarnet() {
 var tab = xmlDoc.getElementsByTagName( "personne" );
 for ( i = 0; i < tab.length; i++ ) {
  alert( tab[ i ].getAttribute( "nom" ) );
 }
}
</script>
<body onload="init()">
</body>
</html>
```

La construction DOM déclenche un événement qui appelle une méthode de traitement (`afficherCarnet`). À des fins de simplifications sur le document produit, nous nous sommes contentés d'afficher, dans une boîte de dialogue, les noms de notre carnet.

JDOM

DOM est ce qu'on pourrait nommer une API portable, c'est-à-dire qu'elle offre un ensemble de méthodes de manipulation XML homogène et donc indépendantes de chaque langage. Cette indépendance a nécessairement l'inconvénient d'ignorer les capacités des langages et de nuire, en quelque sorte, à la productivité du développeur. Le projet Open Source JDOM, pour la plate-forme Java, est parti de cette idée.

Il n'y a tout d'abord pas d'interface pour représenter tous les nœuds, ni un ensemble d'interfaces dérivées pour chaque catégorie. Avec JDOM (*http://www.jdom.org/*), il n'y a que des classes représentant chaque type de nœud (le terme classe doit être compris ici comme une proposition de traitements et pas seulement de signatures de méthodes, comme c'est le cas avec les interfaces).

Lors de la création d'un document, l'API JDOM vérifie toujours qu'une opération a du sens, alors que le caractère générique des nœuds DOM limite les possibilités de contrôle. JDOM facilite également toutes les opérations de lecture et écriture alors que, comme nous l'avons vu, ces opérations, bien que courantes, sont externes à l'API DOM. Comme de nombreuses API Java ne fonctionnent qu'avec DOM, JDOM dispose également de passerelles vers cette représentation. De plus, il est possible de générer des événements SAX à partir d'une arborescence DOM.

Les classes de base

Voici les classes de base disponibles dans le package `org.jdom`. Ces classes sont associées à chaque partie du document XML :

- `Document` : le document ; il faut utiliser la méthode `getRootElement`() pour obtenir la racine ;
- `Element` : un nœud élément ;
- `Attribute` : un nœud attribut ;
- `Text` : un nœud texte ;
- `ProcessingInstruction` : un nœud de traitement ;
- `Namespace` : un nœud espace de noms ;
- `Comment` : un nœud commentaire ;
- `DocType` : un nœud déclaration de type de document ;
- `EntityRef` : un nœud référence d'entité ;
- `CDATA` : un nœud section CDATA.

La comparaison avec DOM

Nous allons présenter ici les avantages de programmer avec JDOM par rapport à DOM. Bien que la productivité avec JDOM soit évidente, comme nous le verrons, il faut cependant garder à l'esprit que de nombreuses API ne sont compatibles qu'avec DOM.

La gestion des éléments

La création d'un élément avec JDOM est directe, puisque chaque nœud est associé à une classe :

```
Element element = new Element("test");
```

Prenons le même exemple avec DOM et l'implémentation standard JAXP : il faut obtenir un document pour pouvoir créer un nœud, ce qui donne :

```
DocumentBuilderFactory factory  = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.newDocument()
Element element = doc.createElement("test");
```

Autre cas, l'ajout d'un texte dans un élément s'écrit avec JDOM :

```
element.setText( "ok" );
```

Ce même ajout avec DOM est réalisé par les instructions :

```
Text t = doc.createTextNode("ok");
element.appendChild( t );
```

Par contre, l'affectation d'une valeur à un attribut est identique entre DOM et JDOM :

```
element.setAttribute( "a", "1" );
```

À noter cependant que dans la version DOM, le nœud devra toujours être converti (cast) en objet `Element`, ce qui n'est pas le cas pour JDOM puisqu'il n'y a pas d'interface générique.

L'ajout d'un élément dans un autre élément suit un processus similaire. Avec JDOM, il s'écrit :

```
element.appendContent( element2 );
```

Alors que dans la version DOM, cela donne :

```
element.appendChild( element2 );
```

La sérialisation de l'arbre

Contrairement à DOM, qui ne dispose pas de solution directe, la sérialisation d'un arbre dans un format texte est disponible simplement avec JDOM et la classe `org.jdom.output.XMLOutputter` :

```
Element e = new Element( "test" );
e.setText( "ok" );
Element ee = new Element( "test2" );
```

```
e.addContent( ee );
XMLOutputter serializer = new XMLOutputter();
Format f = Format.getPrettyFormat();
serializer.setFormat( f );
serializer.output( e, System.out );
```

Ce code produit alors le document suivant :

```
<test>
  ok
  <test2/>
</test>
```

La classe `org.jdom.output.Format` donne des indications sur la présentation du document. Voici les formats disponibles :

- Format compact (`static Format getCompactFormat()`) : il s'agit d'une forme avec normalisation des blancs (retour à la ligne, tabulation…).
- Format indenté (`static Format.getPrettyFormat()`) : il s'agit d'une présentation avec indentation. Lorsque le document peut être modifié par une personne, c'est ce format que l'on utilisera.
- Format brut (`static Format.getRawFormat()`) : il s'agit du document tel qu'il a été analysé ; les blancs sont laissés tels quels.

Cette classe dispose également de quelques options :

- `Format setIndent(java.lang.String indent)` : il s'agit d'un bloc de caractères pour chaque niveau d'indentation.
- `Format setLineSeparator(java.lang.String separator)` : le séparateur de ligne.
- `Format setOmitDeclaration(boolean omitDeclaration)` : intégrer ou non le prologue.
- `Format setOmitEncoding(boolean omitEncoding)` : inclure ou non l'encodage dans le prologue.
- `Format setEncoding(java.lang.String encoding)` : affecter un encodage.
- `Format setTextMode(Format.TextMode mode)` : gestion des blancs avec les constantes `NORMALIZE` (blancs multiples réduits à un blanc), `PRESERVE` (pas de modification) et `TRIM` (suppression de certains blancs).

La gestion des espaces de noms

Les éléments ou les attributs peuvent être construits en spécifiant soit un préfixe et un espace de noms, soit seulement un espace de noms. Voici quelques exemples :

```
Element e = new Element( "test", "p1", "http://www.test.com" );
e.setText( "ok" );
```

```
Element ee = new Element( "test2", "p1", "http://www.test.com" );
ee.addNamespaceDeclaration( Namespace.getNamespace( "p3", "http://www.test3.com" ) );
ee.setAttribute( "a1", "v1", Namespace.getNamespace( "p3", "http://www.test3.com" ) );
e.addContent( ee );
XMLOutputter serializer = new XMLOutputter();
Format f = Format.getPrettyFormat();
serializer.setFormat( f );
serializer.output( e, System.out );
```

Nous obtiendrons alors ce document :

```
<p1:test xmlns:p1="http://www.test.com">
  ok
  <p1:test2 xmlns:p3="http://www.test3.com" p3:a1="v1"/>
</p1:test>
```

Le parsing d'un document XML

Il ne faut pas confondre le parseur et l'API JDOM. JDOM est une représentation mémoire d'un document XML, mais l'étape de construction de cet arbre dépend d'un parseur. JDOM s'appuie sur la couche JAXP et SAX pour être indépendant du parseur avec la classe `org.jdom.input.SAXBuilder`.

Dans l'exemple suivant, un document XML est analysé et sa représentation JDOM est ensuite sérialisée dans la console (sortie standard) :

```
import java.io.File;
import java.io.IOException;
import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;
try {
 SAXBuilder builder = new SAXBuilder();
 Document doc = builder.build(
  new File( "carnet.xml" ) );
 XMLOutputter out = new XMLOutputter( Format.getPrettyFormat() );
 out.output( doc, System.out );
}
catch (JDOMException e) {}
catch (IOException e) {}
```

Nous avons ainsi obtenu :

```
<?xml version="1.0" encoding="UTF-8"?>
<carnet>
 <personne id="p1">
  <nom>Fog</nom>
  <prenom>Phileas</prenom>
```

```
 </personne>
 <personne id="p2">
  <nom>Partout</nom>
  <prenom>Passe</prenom>
 </personne>
</carnet>
```

Le parcours dans l'arborescence JDOM

Plusieurs primitives servent à obtenir les nœuds descendants d'un élément :

- `getChildren` : cette méthode retourne une collection (liste) d'éléments. Contrairement à DOM, tous les autres nœuds sont ignorés.
- `getContent` : cette méthode retourne sous forme d'une collection tous les nœuds possibles (élément, texte). Il revient au développeur de tester ensuite la nature de chaque nœud avant traitement.

Voici un exemple de code affichant tous les éléments d'une arborescence :

```
static void parcours( Element e ) {
 System.out.println( e.getName() );
 List l = e.getChildren();
 for ( int i = 0; i < l.size(); i++ ) {
  parcours( ( Element )l.get( i ) );
 }
}
…
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(
new File( "carnet.xml" ) );
parcours( doc.getRootElement() );
```

Le résultat produit est :

```
carnet
personne
nom
prenom
personne
nom
prenom
```

Nous disposons de méthodes plus fines pour filtrer certains nœuds. La plupart de ces méthodes n'existent pas dans l'API DOM :

- `Element getChild(java.lang.String name)` : retourne l'élément fils correspondant au nom `name`.

- `java.util.List getChildren(java.lang.String name)` : retourne les éléments fils (et non les descendants) qui correspondent au nom `name` (une surcharge avec un espace de noms est également disponible).
- `java.lang.String getChildText(java.lang.String name)` : retourne le texte de l'élément fils correspondant au nom `name`.
- `java.util.Iterator getDescendants()` : liste des éléments descendants.
- `public java.util.Iterator getDescendants(Filter filter)` : liste des éléments descendants filtrés grâce à un objet respectant l'interface `Filter`. Cette interface ne comprend qu'une méthode `matches` qui sera invoquée pour chaque descendant. Elle indiquera, par une valeur booléenne, si l'élément en question devra faire partie du résultat ou non.

Voici maintenant un exemple de code utilisant quelques-unes de ces méthodes :

```
static void parcours( Element e ) {
System.out.println( e.getName() );
List lPersonne = e.getChildren( "personne" );
for ( Iterator it = lPersonne.iterator(); it.hasNext(); ) {
 Element personne = ( Element )it.next();
 System.out.println( "nom = " + personne.getChildText( "nom" ) );
 System.out.println( "prenom = " + personne.getChildText( "prenom" ) );
}
}
```

Nous avons obtenu en résultat l'affichage suivant :

```
carnet
nom = Fog
prenom = Phileas
nom = Partout
prenom = Passe
```

Exercice 6

Parcours d'un document XML

En reprenant le document XML de l'exercice 1, réalisez l'affichage, sous la forme suivante, d'un plan de cours à l'aide de l'API JDOM :

```
Livre:Mon livre
Auteur:nom1 prenom1
Auteur:nom2 prenom2
1. Section1
1.1. Chapitre1
2. Section2
2.1. Chapitre1
2.2. Chapitre2
```

La conversion avec DOM

JDOM possède quelques solutions pour établir une liaison avec DOM, notamment lorsqu'on est en présence d'API incompatible avec JDOM.

La conversion à partir d'un arbre DOM

Un document DOM peut être converti en un document JDOM grâce à la classe `org.jdom.input.DOMBuilder` et la méthode `build`.

Voici un exemple d'utilisation :

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setIgnoringElementContentWhitespace( true );
DocumentBuilder db = dbf.newDocumentBuilder();
Document carnet1 = db.parse( carnet.xml" );
DOMBuilder builder = new DOMBuilder();
org.jdom.Document conversion = builder.build( carnet1 );
org.jdom.Element p1 = conversion.getRootElement().getChild( "personne" );
System.out.println( p1.getAttributeValue( "id" ) );
```

Le document DOM `carnet1` devient un document JDOM avec l'objet `conversion`. Cette conversion peut également s'effectuer sur un élément.

Le passage d'une représentation JDOM vers une représentation DOM peut être réalisé par la classe `org.jdom.output.DOMOutputter` :

```
DOMOutputter outputter = new DOMOutputter();
Document carnet2 = outputter.output( conversion );
```

Il faut faire attention à ces opérations de conversion qui ont nécessairement un coût mémoire (deux arborescences à un moment donné de l'exécution) et un temps de traitement (parcours de tous les nœuds).

Exercice 7

Conversion avec un document DOM

Effectuez un parsing du document de l'exercice 1 pour obtenir un document DOM.

Convertissez ensuite ce document en JDOM.

Ajoutez une section, un chapitre et un paragraphe avec du texte et mettez un titre à ces trois éléments.

Stockez le document produit dans un fichier.

La conversion par événements SAX

Une arborescence JDOM peut être associée à des événements SAX. Nous utilisons la classe `org.jdom.output.SAXOutputter` pour produire ces événements :

```
SAXOutputter so = new SAXOutputter();
so.setContentHandler( new Reader1() );
so.output( convertion );
```

La classe `Reader1` permet d'afficher dans la console les événements au fur et à mesure de leur réception :

```
Locator : org.jdom.output.JDOMLocator@74c3aa
start document
startPrefixMapping :
startElement  carnet carnet org.xml.sax.helpers.AttributesImpl@860d49
characters [
]
startElement  personne personne org.xml.sax.helpers.AttributesImpl@d251a3
characters [
...
```

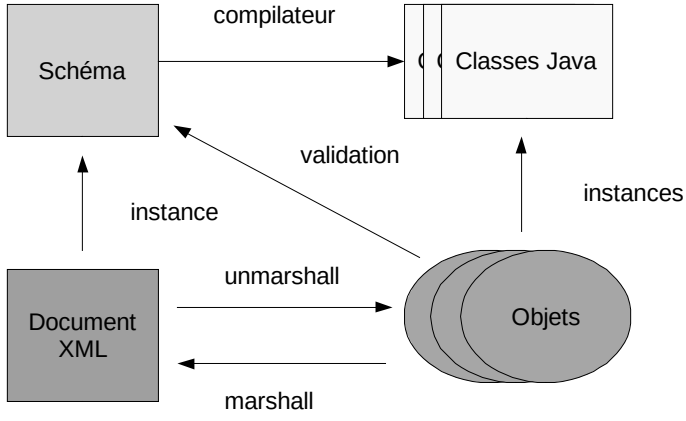
Ce mécanisme présente l'avantage de rendre JDOM exploitable pour des transformations XSLT.

Programmation avec JAXB

JAXB (*Java Architecture for XML Binding*) est une solution à base de correspondance (mapping) entre documents XML et objets. Cette solution est présente depuis la plate-forme 6 et disponible à cette adresse : *https://jaxb.dev.java.net/*.

L'idée est de rendre le document XML transparent pour le développeur, qui ne manipule alors que des objets. Ces objets représentent les éléments et les structures. Ils ne sont pas liés au concept de nœuds comme nous pouvons le trouver dans le modèle DOM.

Figure 8-2
Le modèle JAXB



La figure 8-2 représente le modèle JAXB. Le schéma W3C est la partie centrale de cette architecture, puisque c'est lui qui orchestre la production des classes par une compilation et

la cohérence des objets entre eux (système de validation). La conversion de la représentation objet en représentation XML est appelée *marshalling*, la conversion inverse étant nommée *unmarshalling*.

Le compilateur JAXB

Prenons l'exemple du schéma W3C `carnet.xsd` composé d'éléments `personne` :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="carnet" type="carnetType"/>
<xs:element name="personne" type="personneType"/>
<xs:element name="nom" type="xs:string"/>
<xs:element name="prenom" type="xs:string"/>
<xs:complexType name="carnetType">
<xs:sequence maxOccurs="unbounded">
 <xs:element ref="personne"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="personneType">
<xs:sequence>
 <xs:element ref="nom"/>
 <xs:element ref="prenom"/>
</xs:sequence>
<xs:attribute type="xs:string" name="id"/>
</xs:complexType>
</xs:schema>
```

Le compilateur s'appelle `xjc`. Il a pour principales options :

- `d` : il s'agit du répertoire de destination des classes produites.
- `p` : c'est le package des classes produites.
- `xmlschema` : permet d'utiliser un schéma W3C lors de la génération des classes.
- `relaxng` : permet d'utiliser un schéma RelaxNG lors de la génération des classes (cette option n'était pas opérationnelle au moment de nos tests).
- `dtd` : permet d'utiliser une DTD lors de la génération des classes.

La génération des classes à l'aide du schéma `carnet.xsd` est alors réalisée par la commande suivante :

```
C:\Sun\jwsdp-2.0\jaxb\bin>xjc -p demos.jaxb  -xmlschema -d src\demos\jaxb
xml-data\carnet.xsd
parsing a schema...
compiling a schema...
```

Ce qui donne les fichiers suivants :

```
CarnetType.java
ObjectFactory.java
PersonneType.java
```

L'opération unmarshalling

À l'aide des classes que nous avons obtenues grâce au compilateur, nous allons maintenant pouvoir manipuler le document XML :

```
JAXBContext jc = JAXBContext.newInstance( "demos.jaxb" );
Unmarshaller um = jc.createUnmarshaller();
JAXBElement element = ( JAXBElement )um.unmarshal( new File( "carnet.xml" ) );
CarnetType ct = ( CarnetType )element.getValue();
List<PersonneType> lp = ct.getPersonne();
for ( int i = 0; i < lp.size(); i++ ) {
 PersonneType tp = lp.get( i );
 System.out.println( tp.getNom() );
 System.out.println( tp.getPrenom() );
}
```

L'objet `jc` donne accès aux fonctions de marshalling et unmarshalling. Pour convertir le document XML en objets, nous allons utiliser un objet `Unmarshaller`, qui dispose de la fonction `unmarshall` avec un fichier XML en argument. L'élément racine est obtenu par la méthode `getValue` et de type `CarnetType`. Ce dernier contient la liste de personnes sous la forme d'une collection (`List`). Nous n'avons alors qu'un parcours à effectuer pour afficher les noms et prénoms.

L'opération marshalling

L'opération de marshalling est l'étape de reconstruction du document XML à partir de sa représentation objet :

```
JAXBContext jc = JAXBContext.newInstance( "demos.jaxb" );
…
Marshaller ms = jc.createMarshaller();
ms.setProperty( "jaxb.formatted.output", Boolean.TRUE );
ms.marshal( element, new FileWriter( "c:/carnet.xml" ) );
```

Le fichier produit, grâce à l'objet `element`, peut être indenté à l'aide de la propriété `jaxb.formatted.output`.

Nous disposons également des propriétés suivantes :

- `jaxb.encoding` : l'encodage du document, par défaut `UTF-8`.
- `jaxb.formatted.output` : l'indentation du document, par défaut `false`.
- `jaxb.schemaLocation` : la localisation du schéma lié à un espace de noms ; par défaut aucun.
- `jaxb.noNamespaceSchemaLocation` : la localisation du schéma non lié à un espace de noms ; par défaut aucun.

Exercice 8

Marshalling et unmarshalling

Créez le schéma du document de l'exercice 1 puis générez les classes associées à l'aide du compilateur JAXB. Créez une forme objet (unmarshall) du document XML et affichez la table des matières (sections et chapitres). Enfin, recréez un document XML à partir de la forme précédente (marshalling).

La correspondance entre les types simples des schémas et les types Java

Pour que la conversion XML/objet se déroule convenablement, les types simples par défaut des schémas W3C ont été convertis en types analogues, à partir des types de base Java.

En voici une synthèse :

```
Type dans le schéma : Type Java
xsd:string : java.lang.String
xsd:integer : java.math.BigInteger
xsd:int : int
xsd.long : long
xsd:short : short
xsd:decimal : java.math.BigDecimal
xsd:float : float
xsd:double : double
xsd:boolean : boolean
xsd:byte : byte
xsd:QName : javax.xml.namespace.QName
xsd:dateTime : java.util.Calendar
xsd:base64Binary : byte[]
xsd:hexBinary : byte[]
xsd:unsignedInt : long
xsd:unsignedShort : int
xsd:unsignedByte : short
xsd:time : java.util.Calendar
xsd:date : java.util.Calendar
xsd:anySimpleType : java.lang.String
```

Programmation avec XSLT

Nous allons ici présenter comment réaliser, par programmation, des transformations XSLT.

La technologie JAXP

JAXP est une partie de la plate-forme Java JSE. Outre les opérations de parsing par SAX et DOM, elle dispose également de solutions pour XSLT, tout en bénéficiant de sa technique modulaire, pour remplacer le parseur ou le moteur de transformation.

Les transformations XSLT

Comme nous l'avons vu dans les parties consacrées à SAX et DOM, JAXP est une couche indépendante des parseurs. Cette indépendance est également présente pour les transformations XSLT avec la classe `javax.xml.transform.TransformerFactory`.

Pour réaliser une transformation, il faut disposer d'un document XSLT et d'un document XML. Ces documents peuvent être fournis grâce aux classes suivantes :

- `javax.xml.transform.dom.DOMSource` : le document est fourni par un arbre DOM.
- `javax.xml.transform.sax.SAXSource` : le document est fourni sous forme d'événements SAX.
- `javax.xml.transform.stream.StreamSource` : le document est fourni par un flux ; généralement, nous utiliserons un flux fichier avec un chemin.

De façon analogue, nous aurons, pour le document résultat, des classes équivalentes : `DOMResult`, `SAXResult` et `StreamResult`.

Prenons l'exemple du document `carnet.xml`. Nous allons le transformer à l'aide du document `carnet.xslt` suivant :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/">
 <html>
  <body>
   <xsl:for-each select="//personne">
    <xsl:value-of select="nom"/>
    <xsl:value-of select="prenom"/>
    <br/>
   </xsl:for-each>
  </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

En n'utilisant que des flux fichier, nous avons réalisé le code de transformations suivant :

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class Test1 {
public static void main(String[] args) throws Throwable {
```

```
 TransformerFactory factory = TransformerFactory.newInstance();
 Transformer t = factory.newTransformer(
  new StreamSource(
   "carnet.xslt" ) );
 t.transform(
  new StreamSource(
   "carnet.xml" ),
  new StreamResult(
   "c:/carnet.html" ) );
}
}
```

Le résultat est une page HTML `carnet.html`.

Les transformations à partir d'un arbre DOM

En s'appuyant sur un document à transformer, fourni sous la forme d'un arbre DOM, la transformation peut être obtenue par le code suivant :

```
// Création d'un document DOM
DocumentBuilderFactory factory1 = DocumentBuilderFactory.newInstance();
DocumentBuilder db = factory1.newDocumentBuilder();
Document doc = db.parse( "carnet.xml" );
...
TransformerFactory factory = TransformerFactory.newInstance();
Transformer t = factory.newTransformer( new StreamSource( "carnet.xslt" ) );

// Transformation de l'arbre DOM
t.transform(
new DOMSource(doc ),
 new StreamResult(
 "c:/carnet.html" ) );
```

Les quatre premières lignes du code servent à créer une référence à un document DOM via l'API JAXP (voir la partie intitulée La technologie JAXP et DOM). La transformation est ensuite réalisée à l'aide d'un objet de la classe `DOMSource` en passant en argument au constructeur la référence à l'arborescence DOM.

L'installation d'un nouveau moteur de transformation

Sur le même principe que pour un parseur SAX ou une implémentation DOM, nous disposons de la propriété système `javax.xml.transform.TransformerFactory` pour utiliser un nouveau moteur de transformation compatible JAXP.

Voici un exemple pour utiliser le moteur SAXON :

```
System.setProperty(
"javax.xml.transform.TransformerFactory",
"net.sf.saxon.TransformerFactoryImpl" );
```

Avec Xalan, un autre moteur Open Source, vous devrez utiliser les classes `org.apache.xalan.processor.TransformerFactoryImpl` ou `org.apache.xalan.xsltc.trax.TransformerFactoryImpl`.

Exercice 9

Utilisation de la transformation XSLT avec DOM

Analysez (parsing) le document de l'exercice 1 et créez l'arborescence DOM associée. Ajoutez dans cet arbre un nouveau chapitre dans une section avec un paragraphe. Enfin, réalisez la transformation de cet arbre par XSLT de telle sorte que le plan du livre soit affiché en HTML.

Réaliser des transformations XSLT avec PHP

Pour réaliser des transformations XSLT avec PHP, il faut que le module `php_xsl` soit installé. Si vous avez installé PHP avec l'installateur Windows, vous ne disposerez pas de ce module par défaut : il vous faut utiliser le fichier ZIP (vous pouvez par exemple écraser le répertoire `php` par le contenu de ce fichier).

Voici un exemple de transformation :

```
<html>
<body>
<?
 $doc = new DOMDocument();
 $doc->load( "carnet.xml" );
 $xsl = new XSLTProcessor();
 $xslt = new DOMDocument();
 $xslt->load( "carnet.xslt" );
 $xsl->importStyleSheet( $xslt );
 echo $xsl->transformToXML($doc);
?>
</body>
</html>
```

Comme vous pouvez le constater, nous utilisons deux arborescences DOM : l'une pour le document XML et l'autre pour la feuille de styles. Nous nous sommes contentés de mettre en sortie le résultat de la transformation XSLT.

Réaliser des transformations XSLT avec ASP

Dans le code ASP suivant, nous chargeons le document XML source et la feuille de styles de la même manière, avec l'instruction `load`. La transformation s'effectue par l'instruction `transformNode` qui s'applique à partir du document XML.

```
<%@LANGUAGE="VBSCRIPT" CODEPAGE="1252"%>
<html>
<body>
```

```
<%
Dim doc
Set doc = Server.CreateObject("Microsoft.XMLDOM")
doc.async = false
doc.load( Server.MapPath( "carnet.xml" ) )
if doc.parseError.errorcode <> 0 then
 response.write( "Erreur de parsing " & doc.parseError.reason )
else
 set xslt = Server.CreateObject( "Microsoft.XMLDOM" )
 xslt.async = false
 xslt.load( Server.MapPath( "carnet.xslt" ) )
 if xslt.parseError.errorCode <> 0 then
  response.write( "Erreur de parsing de la feuille de style " & xslt.parseError.reason

 else
  response.write( doc.transformNode( xslt ) )
 end if

end if
%>
</body>
</html>
```

Correction des exercices

Exercice 1

`PlanReader.java` : cette classe gère l'affichage du plan de cours en fonction des événements SAX.

```
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;

public class PlanReader implements ContentHandler {
public void endDocument() throws SAXException {
}
public void startDocument() throws SAXException {
}
public void characters(char[] ch, int start, int length)
  throws SAXException {
}
public void ignorableWhitespace(char[] ch, int start, int length)
 throws SAXException {
}
public void endPrefixMapping(String prefix) throws SAXException {
}
public void skippedEntity(String name) throws SAXException {
}
```

```
public void setDocumentLocator(Locator locator) {
}
public void processingInstruction(String target, String data)
  throws SAXException {
}
public void startPrefixMapping(String prefix, String uri)
  throws SAXException {
}
public void endElement(String namespaceURI, String localName,   String qName)
  throws SAXException {
}

private int numSection = 0;
private int numChapitre = 0;

public void startElement(String namespaceURI, String localName,
String qName, Attributes atts) throws SAXException {
 if ( "livre".equals( localName ) )
  System.out.println(
   "Livre:" + atts.getValue( "titre" ) );
 else
 if ( "auteur".equals( localName ) )
   System.out.println(
    "Auteur:" + atts.getValue( "nom") + " " + atts.getValue( "prenom" ) );
 else
 if ( "section".equals( localName ) ) {
  System.out.println(
   ( ++numSection ) + ". " + atts.getValue( "titre" ) );
        numChapitre = 0;
    }
 else
 if ( "chapitre".equals( localName ) ) {
  System.out.println( numSection + "." + ( ++numChapitre ) + ".
  ➥" + atts.getValue( "titre" ));
}
}
}
```

`PlanLivreSAX.java` : il s'agit de la classe principale. Il existe différentes possibilités pour obtenir un objet de type `XMLReader`.

```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public class PlanLivreSax {

public static void main(String[] args) throws Throwable {
 XMLReader xr = XMLReaderFactory.createXMLReader();
 xr.setContentHandler( new PlanReader() );
 xr.parse( "livre.xml" );
}
}
```

Exercice 2

La classe `ReaderConsole` implémente l'interface `ContentHandler`. Pour simplifier le code, nous avons hérité de la classe `DefaultHandler`, qui implémente cette interface sans traitement.

```
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class ReaderConsole extends DefaultHandler {
public void characters(char[] ch, int start, int length)
 throws SAXException {
 System.out.println( "characters [" + new String( ch, start, length ) + "]" );
}
}
```

La classe `FiltreMaj` applique le filtre par la surcharge de la méthode `characters`.

```
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLFilterImpl;
import org.xml.sax.helpers.XMLReaderFactory;

public class FiltreMaj {

static class EnMajFilter extends XMLFilterImpl {
 public EnMajFilter( XMLReader reader ) {
  super( reader );
 }
 public void characters(
  char[] ch,
  int start,
  int length ) throws SAXException {
  for ( int i = start; i < start + length; i++ ) {
   if ( Character.isLowerCase( ch[ i ] ) )
    ch[ i ] = Character.toUpperCase( ch[ i ] );
  }
  super.characters( ch, start, length );
 }
}
public static void main( String[] args ) throws Exception {
 XMLReader xr = XMLReaderFactory.createXMLReader();
 EnMajFilter filter = new EnMajFilter( xr );
 filter.setContentHandler( new ReaderConsole() );
 filter.parse( "livre.xml" );
}
}
```

Exercice 3

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

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class PlanLivreDOM {

public static void main( String[] args ) throws Throwable {
 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
 dbf.setIgnoringElementContentWhitespace( true );
 DocumentBuilder db = dbf.newDocumentBuilder();
 Document d = db.parse( "livre.xml" );
 Element root = d.getDocumentElement();
 System.out.println( "Livre:" + root.getAttribute( "titre" ) );
 NodeList auteurs = root.getElementsByTagName( "auteur" );
 for ( int i = 0; i < auteurs.getLength(); i++ ) {
  Element a = ( Element )auteurs.item( i );
  System.out.println( "Auteur : " + a.getAttribute( "nom")
  ➥+ " " + a.getAttribute( "prenom" ) );
}
NodeList sections = root.getElementsByTagName( "section" );
for ( int i = 0; i < sections.getLength(); i++ ) {
  Element s = ( Element )sections.item( i );
  System.out.println( ( i + 1 ) + "." + s.getAttribute( "titre" ) );
  NodeList chapitres = s.getElementsByTagName( "chapitre" );
  for ( int j = 0; j < chapitres.getLength(); j++ ) {
   Element c = ( Element )chapitres.item( j );
   System.out.println( ( i + 1 ) + "." + ( j + 2 ) + " " + c.getAttribute( "titre" ) );
        }
    }
}
}
```

Exercice 4

```
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.bootstrap.DOMImplementationRegistry;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

public class Livre {
```

```
private Document d;

public Livre( String titre ) throws Throwable {
    DocumentBuilderFactory db = DocumentBuilderFactory.newInstance();
    d = db.newDocumentBuilder().newDocument();
    Element e = d.createElement( "livre" );
    e.setAttribute( "titre", titre );
    d.appendChild( e );
}

public void ajouterSection( String section ) {
    Element e = d.createElement( "section" );
    e.setAttribute( "titre", section );
    d.getDocumentElement().appendChild( e );
}

public void ajouterChapitre( String section, String chapitre ) {
    Element e = d.createElement( "chapitre" );
    e.setAttribute( "titre", chapitre );
    NodeList nl = d.getElementsByTagName( "section" );
    for ( int i = 0; i < nl.getLength(); i++ ) {
        Element ee = ( Element )nl.item( i );
        if ( section.equals( ee.getAttribute( "titre" ) ) ) {
            ee.appendChild( e );
            break;
        }
    }
}

public void ajouterParagraphe( String chapitre, String texte ) {
    Element e = d.createElement( "paragraphe" );
    e.appendChild( d.createTextNode( texte ) );
    NodeList nl = d.getElementsByTagName( "chapitre" );
    for ( int i = 0; i < nl.getLength(); i++ ) {
        Element ee = ( Element )nl.item( i );
        if ( chapitre.equals( ee.getAttribute( "titre" ) ) ) {
            ee.appendChild( e );
            break;
        }
    }
}

public void afficherLivre() throws Throwable {
    DOMImplementationRegistry registry = DOMImplementationRegistry.newInstance();
    DOMImplementationLS impl =
       (DOMImplementationLS)registry.getDOMImplementation("LS");
    LSSerializer ls = impl.createLSSerializer();
    System.out.println( ls.writeToString( d ) );
}
```

```
/**
 * @param args */
public static void main(String[] args) throws Throwable {
    Livre l = new Livre( "test" );
    l.ajouterSection( "section1" );
    l.ajouterSection( "autre" );
    l.ajouterChapitre( "section1", "ch1" );
    l.ajouterChapitre( "autre", "ch2" );
    l.ajouterParagraphe( "ch1", "ok1" );
    l.ajouterParagraphe( "ch1", "ok2" );
    l.afficherLivre();
}

}
```

Exercice 5

Lors de la fusion, il est important d'utiliser l'instruction `importNode` pour agglomérer les nœuds de documents différents.

```
import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.bootstrap.DOMImplementationRegistry;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

public class Fusion {

public static void main(String[] args) throws Throwable {
 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
 Document dbiblio = dbf.newDocumentBuilder().newDocument();
 Element root1 = dbiblio.createElement( "biblio" );
 dbiblio.appendChild( root1 );

 File dir = new File( args[ 0 ] );
 String[] files = dir.list();
 for ( int i = 0; i < files.length; i++ ) {
  DocumentBuilder db = dbf.newDocumentBuilder();
  Document d = db.parse(
   new File( dir, files[ i ] ) );
  Element root = d.getDocumentElement();
  root1.appendChild( dbiblio.importNode( root, true ) );
 }

DOMImplementationRegistry registry =   DOMImplementationRegistry.newInstance();
 DOMImplementationLS impl =
```

```
  (DOMImplementationLS)registry.getDOMImplementation("LS");
 LSSerializer ls = impl.createLSSerializer();
 System.out.println( ls.writeToString( dbiblio ) );
}
}
```

Exercice 6

```
import java.io.File;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;

public class JDOMLecture {

public static void main(String[] args) throws Throwable {
SAXBuilder sb = new SAXBuilder();
 Document doc = sb.build(
  new File( "livre.xml" ) );
Element livre = doc.getRootElement();
 System.out.println( "Livre " + livre.getAttributeValue( "titre" ) );
 List lauteurs = livre.getChild( "auteurs" ).getChildren( "auteur" );
 for ( int i = 0; i < lauteurs.size(); i++ ) {
  Element ae = ( Element )lauteurs.get( i );
  System.out.println( "Auteur:" + ae.getAttributeValue( "nom" )
  ➥+ " " + ae.getAttributeValue( "prenom" ) );
          }
  List lsections = livre.getChild( "sections" ).getChildren( "section" );
  for ( int i = 0; i < lsections.size(); i++ ) {
   Element section = ( Element )lsections.get( i );
   System.out.println( ( i + 1 ) + ". " + section.getAttributeValue( "titre" ) );
   List lchapitres = section.getChildren();
   for ( int j = 0; j < lchapitres.size(); j++ ) {
    Element chapitre = ( Element )lchapitres.get( j );
    System.out.println( ( i + 1 ) + "." + ( j + 1 ) + ". "
    ➥+ chapitre.getAttributeValue( "titre" ) );
   }
 }
}
}
```

Exercice 7

```
import java.io.FileOutputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.jdom.Element;
```

```
import org.jdom.input.DOMBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;
import org.w3c.dom.Document;

public class DOM2JDOM {


public static void main( String[] args ) throws Throwable {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setIgnoringElementContentWhitespace( true );
    DocumentBuilder db = dbf.newDocumentBuilder();
    Document d = db.parse(
    "livre.xml" );

    DOMBuilder builder = new DOMBuilder();
    org.jdom.Document conversion = builder.build( d );

    // Création section
    Element section = new Element( "section" );
    section.setAttribute( "titre", "section3" );
    Element ch = new Element( "chapitre" );
    ch.setAttribute( "titre", "chapitreZ" );
    Element p = new Element( "paragraphe" );
    p.setText( "Mon nouveau paragraphe..." );
    ch.addContent( p );
    section.addContent( ch );

    conversion.getRootElement().getChild( "sections" ).addContent( section );

    Format f = Format.getPrettyFormat();
    f.setEncoding( "iso-8859-1" );
    XMLOutputter serializer = new XMLOutputter();
    serializer.setFormat( f );
    serializer.output( conversion, new FileOutputStream( " livre2.xml" ) );
}
}
```

Exercice 8

```
import java.io.File;
import java.io.FileWriter;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;

public class ExoMain {
```

```
public static void main(String[] args) throws Throwable {
JAXBContext jc = JAXBContext.newInstance( "exo.xdj.jaxb" );
Unmarshaller um = jc.createUnmarshaller();
JAXBElement<TypeLivre> element = ( JAXBElement<TypeLivre> )um.unmarshal(
 new StreamSource(
  " livre.xml" ), TypeLivre.class );
 TypeLivre tl = element.getValue();
 TypeSections ts = tl.getSections();
 List<TypeSection> l = ts.getSection();
 for ( int i = 0; i < l.size(); i++ ) {
  TypeSection sect = l.get( i );
  System.out.println( ( i + 1 ) + ". " + sect.getTitre() );
  List<TypeChapitre> l2 = sect.getChapitre();
  for ( int j = 0; j < l2.size(); j++ ) {
   System.out.println( ( i + 1 ) + "." + ( j + 1 ) + " " + l2.get( j ).getTitre() );
 }
}
Marshaller ms = jc.createMarshaller();
ms.setProperty( "jaxb.formatted.output", Boolean.TRUE );
ms.marshal( element, new FileWriter( " livre2.xml" ) );
}
}
```

Exercice 9

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

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Test1 {
public static void main(String[] args) throws Throwable {

DocumentBuilderFactory factory1 = DocumentBuilderFactory.newInstance();
 DocumentBuilder db = factory1.newDocumentBuilder();
 Document doc = db.parse( "livre1.xml" );

Element section = doc.createElement( "section" );
 section.setAttribute( "titre", "titre1" );
 Element chapitre = doc.createElement( "chapitre" );
 chapitre.setAttribute( "titre", "titre1" );
 Element paragraphe = doc.createElement( "paragraphe" );
```

```
 paragraphe.appendChild( doc.createTextNode( "Test" ) );
 section.appendChild( chapitre );
 chapitre.appendChild( paragraphe );
 doc.getElementsByTagName( "sections" ).item( 0 ).appendChild( section );

 TransformerFactory factory = TransformerFactory.newInstance();
 Transformer t = factory.newTransformer(
  new StreamSource( “livre.xsl" ) );
t.transform(
  new DOMSource(
   doc ),
  new StreamResult(
  "c:/test.html" ) );
}
}
```

Nous avons employé la feuille de styles suivante pour produire le plan :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/livre">
 <html>
  <body>
   <xsl:apply-templates select="sections" mode="TM"/>
   <xsl:apply-templates select="sections" mode="FULL"/>
  </body>
 </html>
</xsl:template>

<xsl:template match="sections" mode="TM">
 <xsl:for-each select="section">
  <a href="#{@titre}">
   <xsl:number/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@titre"/>
  </a>
  <br />
  <xsl:for-each select="chapitre">
   <a href="#{@titre}">
    <xsl:number count="section|chapitre" level="multiple"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@titre"/>
   </a>
   <br />
  </xsl:for-each>
 </xsl:for-each>
</xsl:template>
```

```
<xsl:template match="sections" mode="FULL">
 <xsl:for-each select="section">
  <a name="{@titre}">
   <h1>
   <xsl:number/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@titre"/>
  </h1>
 </a>
 <xsl:for-each select="chapitre">
  <a name="{@titre}">
   <h2>
    <xsl:number count="section|chapitre" level="multiple"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@titre"/>
   </h2>
  </a>
  <xsl:apply-templates select="paragraphe"/>
 </xsl:for-each>
</xsl:for-each>
</xsl:template>

<xsl:template match="paragraphe">
<p>
 <xsl:value-of select="."/>
</p>
</xsl:template>

</xsl:stylesheet>
```

Index