*Chapitre XIII*

# *Analyser un document XML - Les Parsers*

Un analyseur XML (*parser* en anglais) a pour rôle d'analyser un document XML et de servir de lien avec une application de traitement. Il a pour objectif de parcourir un document (static) XML (i.e. lire un fichier), pour en analyser (de manière dynamique) le contenu et faire un traitement.

Le but de ce chapitre est de donner un bagage minimal pour programmer avec XML. La programmation permettant d'utiliser les technologies XML (XML, XSD, XSLT, XPath) 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, etc.). 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. A l'inverse, d'autres systèmes pourront eux-même alimenter votre application XML qui devient alors un pont commun entre différents systèmes.[1]

Un analyseur XML (parser) permet de lire des données XML, d'identifier les balises et de passer les données sous forme d'informations directement utilisables par une application. Un analyseur XML sait donc lire un fichier texte écrit en XML et connaît les règles XML (points clés). Par contre, il doit être généraliste et ne connaît pas un format en particulier (i.e. in le connaît pas le schema, la grammaire d'un document particulier).

Les parsers sont à la base d'une autre technologie que nous verrons par la suite : la sérialisation.
Il existe 3 stratégies possibles pour les analyseurs XML (cf Figure XIII .1):

1. transformer toutes les données XML en une structure arborescente en mémoire, un algorithme peut ensuite parcourir cet arbre (DOM)
2. utiliser des événements pendant la lecture du flux (i.e. un automate lit le document). Dans ce cas, soit
   (a) l'automate fonctionne automatiquement (*push*) : le données sont poussées vers l'application (SAX en Java)
   (b) l'automate attend les instructions (*pull*) : l'application va chercher les données (StAX en Java)

[1] extrait de *XML, Cours et exercices*, Alexandre Brillant, editions Eyrolles.

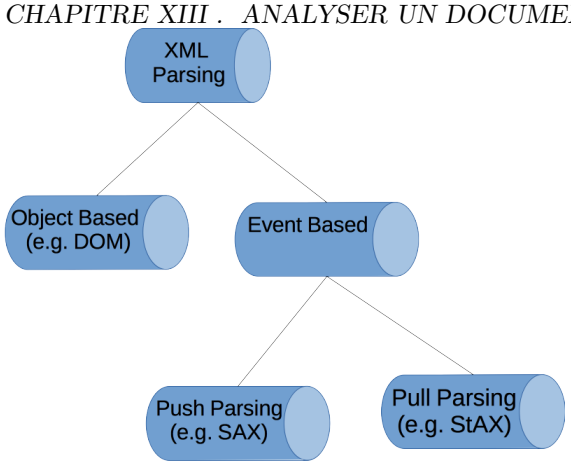


Figure XIII .1: Vue d'ensemble des analyseurs XML (extrait de: The Evolution of JAXP JavaOne 2006 Rahul Srivastava, Apache).

Ces stratégies d'analyse de document XML sont traduites en API (*Application Programming Interface*). Dans ce cadre, une API est une interface de programmation standardisée, indépendante du langage dans laquelle elle est implémentée. Par exemple, en TP/Projet, nous utiliserons l'API DOM via ses implémentations en Java, en javascript, en .Net C#, et en C++ mais il en existe également des implémentations en PHP, ECMAScript, etc. L'avantage est que les concepts et les noms des méthodes sont les même, indépendamment du langage utilisé. Toutes les API n'implémentent pas forcément exactement les mêmes fonctions ni même le même fonctionnement, mais la logique de fonctionnement reste la même. Dans la suite de ce chapitre, nous présenterons des technologies de l'API JAva et d'autres de l'API .Net C#. Le langage sera précisé systématiquement.

En résumé, dans cette longue partie, vous trouverez donc les sections suivantes :

- **Document Object Model: DOM [Java et C#]** qui décrit comment utiliser l'API DOM et la combiner avec XPath pour naviguer dans un document et l'éditer.
- **Forward parsers [Java et C#]** incluant les parsers Sax et Stax en Java, et XmlReader en C#
- **Quelques usages des parsers en C#**, section décrivant quelques usages incluant la génération de schémas, la validation de schémas, les transformations XSLT.

# 1 *Document Object Model*: DOM [Java et C#]

L'API DOM est particulièrement standardisée. Autrement dit, ce qui est dit en Java sera valable dans un autre langage (C#, Javascript ...).

## 1.1 Principe

DOM (*Document Object*[2] *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, attribut, etc.). DOM est une API disponible dans la plupart des langages et maintenue par le W3C.

DOM lit d'abord entièrement le document (sérialisé sous forme de fichier), et construit une structure arborescente. Il traduit ensuite cette représentation en objets (par exemple instances de classes en Java). Cette dernière représentation permet de manipuler/interagir de manière dynamique avec les différents éléments du XML. On peut ainsi accéder directement à la structure des documents ainsi qu'à son contenu.

L'exemple des figures XIII .2, XIII .3 et XIII .4 représente un même document XML sous 3 formes différentes. La dernière est la représentation en mémoire du DOM du document.

[2] *Objet* comme dans Programmation Orientée Objet.

Forme sérialisée

```
<?xml version="1.0" encoding="UTF-8" ?>
<patient id="20060105">
    <nom>Anna Patho</nom>
    <dateNaissance>1914-02-01</dateNaissance>
</patient>
```

Figure XIII .2: Format d'un fichier. Facilite le stockage et la transmission / l'échange des données..

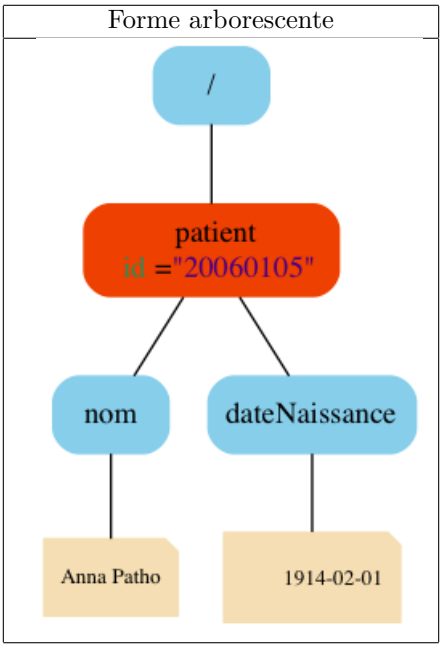


Figure XIII .3: Représentation en arbre, permet de comprendre, de manipuler intellectuellement le document (forme et contenu).

La représentation DOM (simplifiée) de la figure XIII .4 permet de retrouver les relations entre les différentes structures XML grâce à des références. On conserve en effet l'organisation: *firstChild* correspond au premier sous-arbre, *attributes* correspond aux attributs d'un nœud, *nextSibling* correspond au nœud suivant d'un même niveau, etc. On a, dans ce schéma, 4 grands types de structure: *Document*, *Element*, *Attribute*, *Text*, etc. La valeur d'un nœud est donné par *nodeValue*. On peut noter que la valeur d'un nœud *Text* est son texte, mais que pour un élément, sa valeur est `NULL`.

## 1.2 Anatomie d'un application utilisant DOM

Nous allons prendre ici l'exemple d'une application Java utilisant DOM (cf Figure XIII .6). Les applications dans les autres langages se comportent de manière similaire.

La méthode contenant des instructions DOM doit d'abord instancier un *parser*. Ce *parser* DOM lit alors le document et crée une arborescence DOM (instruction `parser.parse("cheminJusquauDocument.xml");`). L'application récupère alors l'arbre DOM sous forme de référence vers un objet de type `Document`. On peut alors appliquer des instructions DOM sur cette référence.

## 1.3 Principales méthodes utilisées

L'API DOM définit une *interface*. Cette interface définit la signature des méthodes que devront être implémentées dans les classent qui *implémentent* cette *interface*. L'API DOM définit également différentes classes ainsi que des méthodes associées. Nous notons ici les plus couramment utilisées. Un diagramme UML très simplifié de cette API est représenté figure XIII .7.

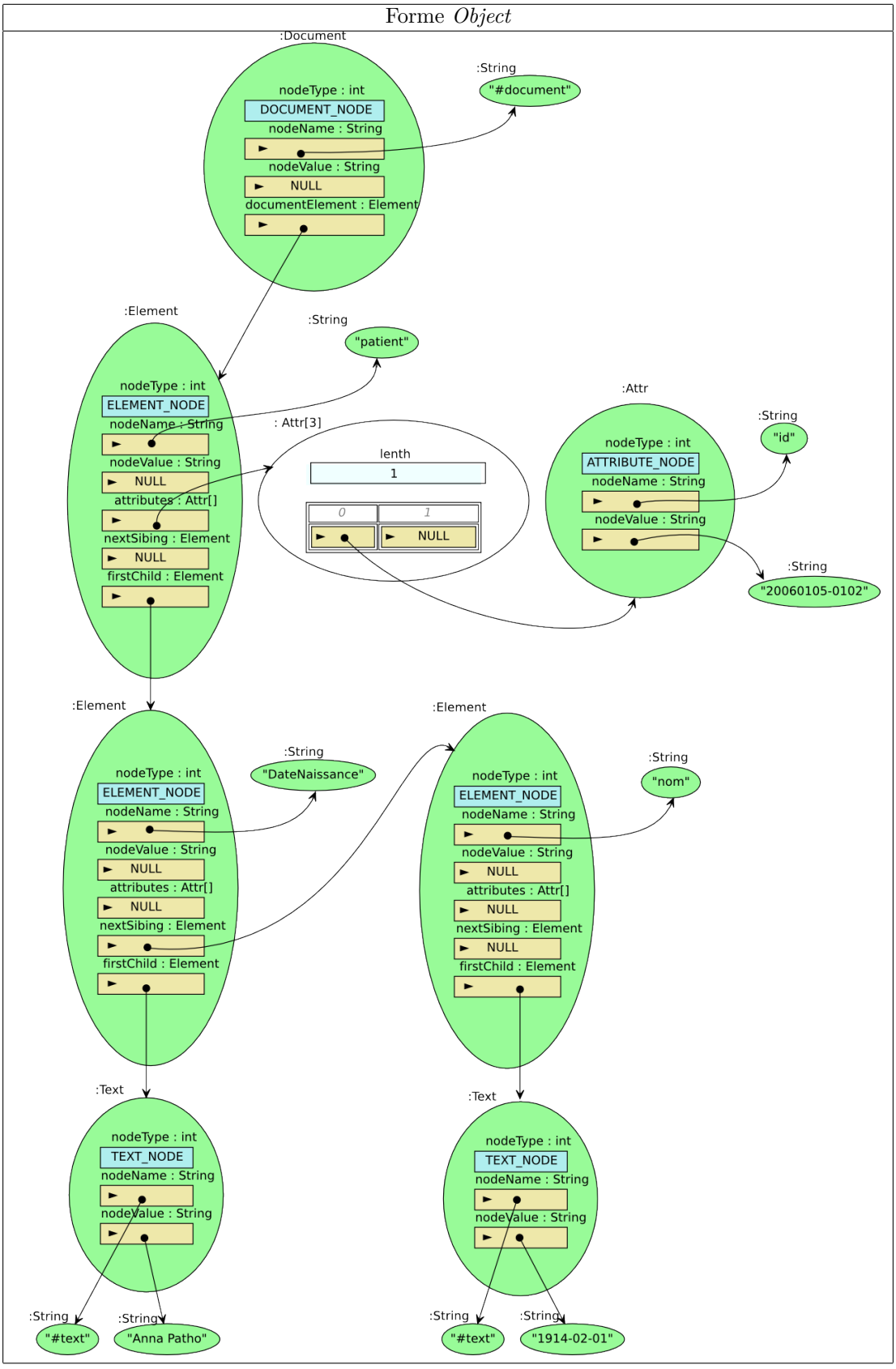


Figure XIII .4: Représentation sous forme d'objets organisés dans la mémoire: permet d'accéder aux éléments DOM via un programme (généralement orienté objet)..

```java
// ici, l'on utilise le parser DOM de Xerces,
//  nous verrons qu'il en existe d'autres en Java
import com.sun.org.apache.xerces.internal.parsers.DOMParser;
import org.w3c.dom.Document;

public class Test {
        public static void main(String[] args) {
                DOMParser parser = new DOMParser();
                parser.parse("doc.xml");
                Document doc = parser.getDocument();

                //------------ instructions DOM ------------
        }
}
```

Figure XIII .5: Anatomie d'une application Java utilisant DOM

```
    public Rectangle(String filename) {
        // charge le document et récupère l'élément racine
        XmlDocument doc = new XmlDocument();
        doc.Load(filename);
        XmlNode root = doc.DocumentElement;
        // Add the namespace.
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(doc.NameTable);
        nsmgr.AddNamespace("", "http://www.univ-grenoble-alpes.fr/rectangle");
        X = Convert.ToDouble(((XmlElement)root).GetElementsByTagName("x").Item
            (0).InnerXml);
        Y = Convert.ToDouble(((XmlElement)root).GetElementsByTagName("y").Item
            (0).InnerXml);
        _width = Convert.ToDouble(((XmlElement)root).GetElementsByTagName("
            width").Item(0).InnerXml);
        _height = Convert.ToDouble(((XmlElement)root).GetElementsByTagName("
            height").Item(0).InnerXml);
    }
```

Figure XIII .6: Ajout à la classe Rectangle d'un constructeur utilisant DOM en C#

### Node (Java) / XmlNode (C#)

Il s'agit d'un type générique de nœud qui regroupe les méthodes communes à tous les nœuds quels que soient leurs types.

Cette interface définie 3 types de méthodes: les méthodes qui donnent des résultats différents en fonction des nœuds, les opérations de lecture de l'arbre DOM et les opérations de modification de l'arbre DOM.

La première catégorie de méthode dépendantes du type de nœud contient les méthodes (ou propriétés en C#) suivantes:

- En Java: `getNodeType() : int` ou en C#: `NodeType : XmlNodeType` qui permet de distinguer la nature du nœud manipulé[3]. Les valeurs entières retournées par cette méthode correspondent à une énumération. Parmi cette énumération, les éléments que nous utiliserons le plus souvent sont (en Java) :
  - `Node.ATTRIBUTE_NODE`
  - `Node.CDATA_SECTION_NODE`
  - `Node.COMMENT_NODE`

[3]Nous verrons plus tard que c'est redondant avec les capacités de certains langages comme Java qui proposent des méthodes d'introspection de type (comme la méthode `instanceof` en Java ou la méthode `typeof` en C#), mais cela est dû au fait que DOM est une API multilangages.

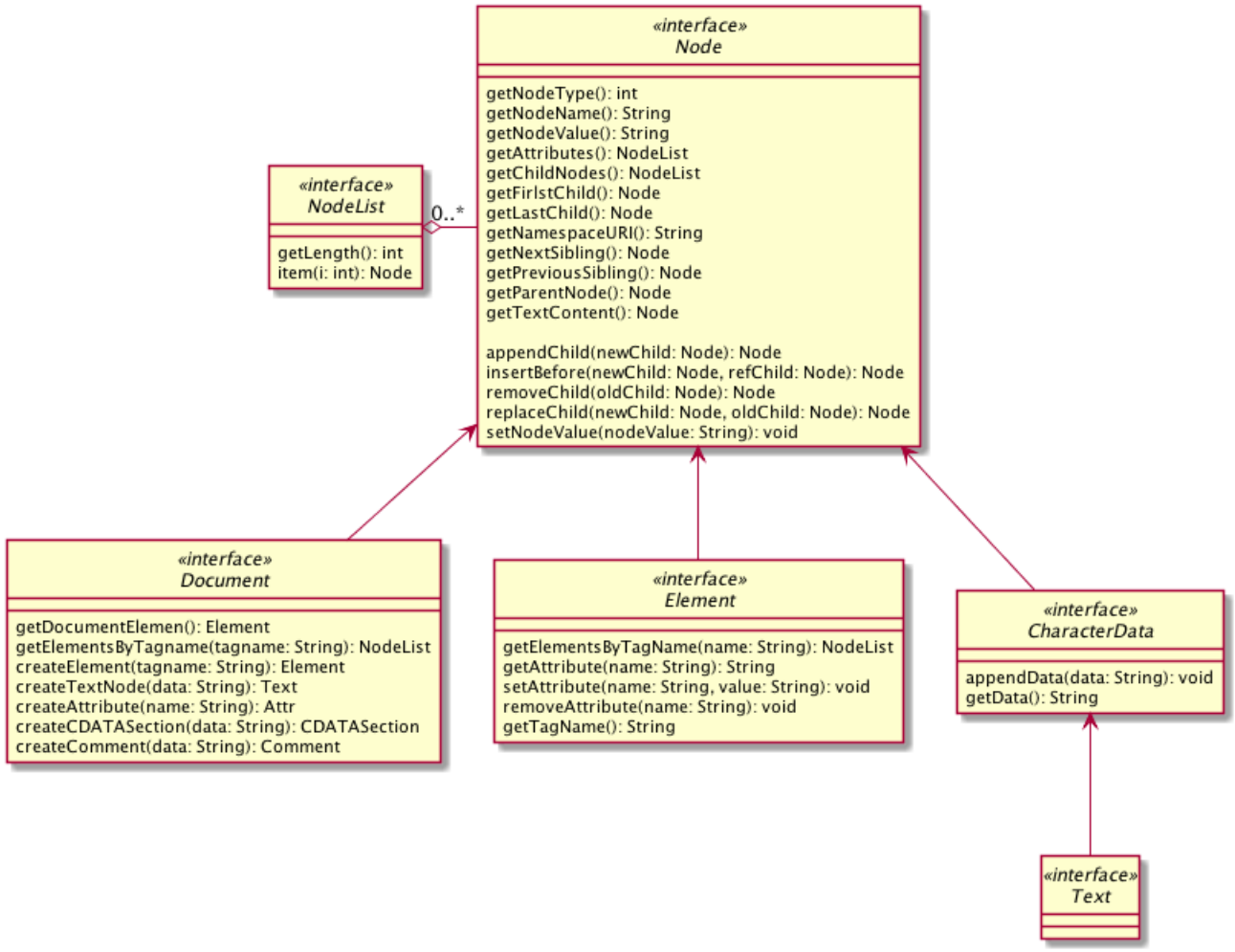


Figure XIII .7: Diagramme UML simplifié de l'API DOM (en Java)

  - `Node.DOCUMENT_NODE`
  - `Node.ELEMENT_NODE`
  - `Node.TEXT_NODE`

  et en C# :

  - `IXMLDOMNodeType.NODE_ELEMENT`
  - `IXMLDOMNodeType.NODE_ATTRIBUTE`
  - `IXMLDOMNodeType.NODE_TEXT`
  - `IXMLDOMNodeType.NODE_CDATA_SECTION`
  - `IXMLDOMNodeType.NODE_COMMENT`
  - `IXMLDOMNodeType.NODE_DOCUMENT`

- En Java: `getNodeName() : String` ou en C#: `Name : String` qui renvoie le nom du nœud. S'il s'agit d'un élément, cela va correspondre au nom de 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. Par exemple `getNodeName() / NodeType` appliqué sur un objet de type `Document` renverra `#Document` ; appliqué sur un objet de type `Text` renverra `#text`.
- En Java: `getNodeValue() : String` ou en C#: `Value : String` : si le nœud est un attribut, cela correspondra à la valeur de l'attribut. Pour toutes les formes de nœud texte (texte simple, section CDATA, etc.), cela correspondra au texte lié. A noter qu'en C# ceci concerne comme cela

a été dit les objets de type Node. En C# la propriété `Value: String` existe aussi mais renvoie `null`. Il est cependant possible en C# de récupérer directement la valeur du texte contenu dans un élément avec la propriété `InnerText` qui concatène tout le texte contenu dans l'élément et ses sous éléments. Il existe également une méthode `InnerXml` qui renvoie le contenu (texte et sous-élements) d'un élément ; si ce contenu n'est que du texte, alors c'est équivalent à `InnerText`.

- En Java: `getAttributes() : NodeList` ou en C#: `Attributes : XmlAttributeCollection?` (on remarquera que cette propriété est de type nullable) : cette méthode ne concerne en fait que les `Element` car seuls les noeuds de type élément contiennent des attributs. En Java la méthode renverra une liste de `Node` contenant les attributs de l'élément. Pour les autres types de nœud que `Element`, cette méthode n'aura pas d'effet. En C#, le type `XmlAttributeCollection` étend le type `XmlNamedNodeMap` qui contient donc une map de nodes. Pour accéder au contenu du premier attribut de la racine par exemple, il faudra faire `root.Attributes.Item(0).InnerText` ; pour tous les afficher :

```
XmlNode root = doc.DocumentElement;
foreach (XmlAttribute item in root.Attributes) {
            Console.WriteLine(item.Name);
            Console.WriteLine(item.InnerText);
        }
```

Les opérations de lecture de l'arbre DOM les plus courantes sont les suivantes:

- En Java `getChildNodes() : NodeList` ou en C# `ChildNodes : XmlNodeList` renvoie la liste des nœuds fils du nœud courant. Cette liste peut être vide (dans le cas des nœuds `Text` par exemple).

Comme DOM ne connaît pas les schémas des documents qu'il traite, il considère que tous les éléments sont **mixed**, c'est-à-dire qu'ils peuvent contenir des sous-éléments **et** du texte. Par conséquent, le premier `Node` fils d'un élément qui contient des sous éléments est très souvent un `Node` de type `Text` qui contient la chaîne de caractère de retour à la ligne et d'indentation.

- En Java `getFirstChild() : Node` ou en C# `FirstChild : XmlNode` : le premier nœud fils
- En Java `getLastChild() : Node` ou en C# `LastChild : XmlNode` : le dernier nœud fils
- En Java `getNamespaceURI : String` ou en C# `NamespaceURI : String` l'URI de l'espace de nom s'il y en a un.
- En Java `getNextSibling() / getPreviousSibling() : Node` ou en C# `NextSibling / PreviousSibling : XmlNode` le nœud adjacent (avant / après)
- En Java `getParentNode() : Node` ou en C# `ParentNode : XmlNode`: le nœud parent
- En Java `getTextContent() : String` ou en C# `InnerText : String` : la concaténation des textes contenus dans le nœud (sans les balises des sous-éléments).

Pour modifier l'arbre DOM, on utilise généralement:

- En Java `appendChild(newChild : Node) : Node` ou en C# `AppendChild(newChild : XmlNode) : XmlNode` : ajoute un fils au nœud courant (après les autres)
- En Java `insertBefore/insertAfter(newChild: Node, refChild: Node): Node` ou en C# `InsertBefore/InsertAfter(newChild: XmlNode, refChild: XmlNode): XmlNode` : insert un nouveau fils avant/après le fils `refChild`
- En Java `removeChild(oldChild: Node) : Node` ou en C# `RemoveChild(oldChild: XmlNode) : XmlNode`: supprime le fils `oldChild` de la liste des nœuds fils

- En Java `replaceChild(newChild: Node, oldChild: Node): Node` ou en C# `ReplaceChild(newChild: XmlNode, oldChild: XmlNode): XmlNode` : remplace un nœud fils par un nouveau.

- En Java `setNodeValue(nodeValue: String): void` ou en C# `Value String` : modifie la valeur du nœud. Cette opération va dépendre du type de nœud.

Les classes suivantes implémentent toutes l'interface `Node` (Java) / `XmlNode` (C#). Elles contiennent donc toutes les méthodes précédentes, plus d'autres méthodes.

### Document (Java) / XmlDocument (C#)

Un objet de type `Document` (Java) / `XmlDocument` (C#) représente l'ensemble du document. Il contient une référence vers l'`Element` (Java) / `XmlElement` (C#) racine. Il sert également à construire différents types de nœuds. Chaque nœud construit appartient à ce document. Les méthodes spécifique à `Document` que nous utiliserons le plus souvent sont:

- `getDocumentElement() : Element` (Java) ou `DocumentElement : Element` (C#) renvoie l'élément racine (c'est-à-dire le premier élément du document).

- `getElementsByTagName(tagname: String) : NodeList` (Java) ou `GetElementsByTagName(tagname: String) : XmlNodeList` (C#) retourne la liste des éléments ayant pour nom celui passé en paramètre.

- `createElement(tagname: String) : Element` (Java) ou `CreateElement(tagname: String) : XmlElement` (C#) crée un nouvel élément qui appartient à ce document

- `createTextNode(data: String)` (Java) ou `CreateTextNode(data: String)` (C#) crée un nouveau nœud texte

- `createAttribute(name: String)` (Java) ou `CreateAttribute(name: String)` (C#) crée un nouveau nœud attribut.

- `createCDATASection(data: String)` (Java) ou `CreateCDATASection(data: String)` (C#) : crée une section CDATA

- `createComment(data: String)` (Java) ou `CreateComment(data: String)` (C#) : crée un nouveau nœud commentaire

### Element (Java) / XmlElement (C#)

L'`Element` (Java) / `XmlElement` (C#) [NB: à partir de maintenant, quand c'est possible, on écrira de manière plus compacte ceci : `Element/XmlElement`, sous entendu, pour les langages Java/C#] est probablement le nœud le plus employé et caractérise un élément XML (balise ouvrante, contenu (ou non) et balise fermante). Les méthodes particulière à `Element` (Java) / `XmlElement` (C#) que nous utiliserons le plus souvent sont les suivantes:

- `getElementsByTagName(name: String): NodeList` (Java) / `GetElementsByTagName(name: String): XmlNodeList` (C#) [NB: à partir de maintenant, on écrira ceci de manière plus compacte lorsque c'est possible : `g/GetElementsByTagName(name: String): NodeList/XmlNodeList` , sous entendu pour les langages Java/C#] renvoie la liste des éléments descendants de l'élément courant qui ont pour nom celui passé en attribut.

On peut noter que la méthode `g/GetElementsByTagName` existe dans `Document/XmlDocument` et `Element/XmlElement`, mais pas dans l'interface `Node/XmlNode`. Or elle renvoie une liste de `Node/XmlNode`. Si l'on considère les 2 instructions suivantes (code Java):

```
Element liste;
liste.getElementsByTagName("livre").item(0).getElementsByTagName("titre");
```

la seconde n'est pas possible puisque `liste.getElementsByTagName("livre").item(0)` est de type `Node` et non de type `Element`. Il faut donc passer par un transtypage (*cast*) explicite:

```
Element livre1 = (Element) liste.getElementsByTagName("livre").item(0);
livre1.getElementsByTagName("titre"); // on a ici à nouveau une liste de Node
```

La même chose doit être faite dans n'importe quel langage utilisant l'API.

- `g/GetAttribute(name: String): String` retourne la valeur de l'attribut dont le nom est passé en paramètre.
- `s/SetAttribute(name: String, valeur: String): void` modifie la valeur de l'attribut.
- `r/RemoveAttribute(name: String): void` supprime l'attribut dont le nom est passé en paramètre
- `getTagName(): String` (Java) ou `Name: String` (C#) renvoie le nom de l'élément.

Pour la gestion des espaces de noms :

- en Java il suffit d'ajouter `NS` au nom de la méthode puis de préciser l'espace de nom. Par exemple, la suppression d'un attribut dans un espace de noms correspond à la méthode `removeAttributeNS(namespaceURI: String, locaName: String)`.
- , les méthodes sont déclinées sous 2 version dont l'une prend en paramètre l'URI du namespace. Par exemple :

```
1            String nsURI = root.NamespaceURI;
2          ((XmlElement) root).RemoveAttribute("xmlns",nsURI);
```

### Text (Java) / XmlText (C#)

Un nœud `Text` (Java) / `XmlText` (C#) contient le texte des éléments. Il est fils d'un `Element/XmlElement`. Les méthodes courantes que nous utiliserons sont:

- `appendData(data: String): void` (Java) ou en C# l'usage `Value : String` et des méthodes et opérateurs de `String` ajoute un bloc de texte à la fin du texte courant.
- `getData(): String` (Java) ou `Value` (C#) renvoie le texte contenu dans le nœud

En Java cette méthode renvoie la même chose que la méthode `getValue()` appliquée à un nœud `Text`. En C# c'est équivalent à utiliser la propriété `InnerText` mais pas `Value`.

- `setData(data: String): void` (Java) ou `Value : String` (C#) écrit le texte passé en paramètre

Exemple en C# (on récupère la coordonnée x du fichier Rectangle.xml):

```
1    XmlElement rootElt = (XmlElement)root;
2    XmlNode xNode = rootElt.GetElementsByTagName("x").Item(0);
3    Console.WriteLine(xNode.InnerXml);
4    XmlText xTextNode = (XmlText) xNode.ChildNodes.Item(0);
5    Console.WriteLine(xTextNode.Value);
```

## 1.4 Utiliser DOM avec XPath

Tout l'intérêt de XPath, c'est justement de pouvoir être utilisé avec une API qui "connait" la totalité de l'arbre XML ... et c'est le cas de DOM ! Nous avons vu que la méthode `g/GetElementsByTagName` permet de récuperer des `NodeList/XmlNodeList` avec le nom d'un élément. Mais pour effectuer des opérations plus complexes comme nous en avons vu en XPath, soit on doit écrire du code Java/C# pour obtenir ce que l'on veut, soit ... on utilise directement XPath.

**En Java.**

... on doit créer une expression XPath, expression que l'on construit à partir d'une fabrique. C'est l'objet instancié de type `XPath` qui évalue une expression écrite sous la forme d'une String et que l'on applique au document. Voici un exemple complet d'une classe, `DOM2XPath`, qui utilise DOM pour parser un document à la construction (ici le document `Bagagerie.xml` du chapitre dédié à XPath) et qui dispose d'une méthode `getXPath(expression: String): NodeList` qui permet d'obtenir une `NodeList` à partir d'une expression XPath :

```
1  package bagages;
2
3  import java.io.IOException;
4  import javax.xml.parsers.DocumentBuilder;
5  import javax.xml.parsers.DocumentBuilderFactory;
6  import javax.xml.parsers.ParserConfigurationException;
7  import javax.xml.xpath.XPath;
8  import javax.xml.xpath.XPathConstants;
9  import javax.xml.xpath.XPathExpressionException;
10 import javax.xml.xpath.XPathFactory;
11 import org.w3c.dom.Document;
12 import org.w3c.dom.NodeList;
13 import org.xml.sax.SAXException;
14
15 public class DOM2XPath {
16
17     Document doc;
18
19     public DOM2XPath(String filename) throws SAXException, IOException {
20         try {
21             DocumentBuilderFactory fabrique = DocumentBuilderFactory.
                   newInstance();
22             DocumentBuilder dBuilder = fabrique.newDocumentBuilder();
23             doc = dBuilder.parse(filename);
24         } catch (ParserConfigurationException e) {
25             System.out.print("ParserConfigurationException  :" + e.getMessage()
                   );
26         } catch (SAXException e) {
27             System.out.print("SAXException  :" + e.getMessage());
28         } catch (IOException e) {
29             System.out.print("IOException  :" + e.getMessage());
30         }
31     }
32
33     public NodeList getXPath(String expression) {
34         XPath xPath = XPathFactory.newInstance().newXPath();
35         try {
36             NodeList nodeList = (NodeList) xPath.compile(expression).evaluate(
                   doc, XPathConstants.NODESET);
37             return nodeList;
38         } catch (XPathExpressionException e) {
39             System.out.println("Erreur dans le XPath");
40         }
41         return null;
42     }
43
```

```
}
```

... et son usage :

```
package bagages;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

 public class GestionBagages {

    public static void main(String[] args) {
        try {
            // --- On parse le document à la construction ---
            DOM2XPath b = new DOM2XPath("src/bagages/bagagerieHF.xml");
            // --- On souhaite récupérer les identifiants de passagers tels que
                 leurs bagages sont des skis ---
            NodeList bagagesNL = (NodeList) b.getXPath("//bagage[description='
                skis']/@idPassager");
            // --- On affiche ces identifiants ---
            for (int i=0; i<bagagesNL.getLength(); i++)
                System.out.println(bagagesNL.item(i).getNodeValue());
        } catch (SAXException | IOException ex) {
            Logger.getLogger(GestionBagages.class.getName()).log(Level.SEVERE,
                null, ex);
        }
    }

}
```

**En C#.**

...

En C# il y a 2 moyens d'utiliser XPath avec des documents DOM : soit via des documents de type `XmlDocument` (comme vu précédemment) qui sont utilisables en lecture et écriture, soit via des documents de type `XpathDocument` qui ne sont accessibles qu'en lecture. Je vous invite à lire ce lien et ses sous-sections pour plus de détails.

Dans ce cours, nous ne considèrerons que le cas des `XmlDocument`. Dans ce cas, les `XmlNode` disposent de 2 méthodes permettant de sélectionner soit 1 seul `XmlNode`, la méthode `SelectSingleNode(expression: String, ns: XmlNamespaceManager):XmlNode`, soit une `XmlNodeList`, la méthode `SelectNodes(expression: String, ns: XmlNamespaceManager) : XmlNodeList`. Ces méthodes requièrent l'usage d'un préfixe associé à une URI dans les expressions utilisées et ce ... même si le document XML n'est pas préfixé ! (pour en savoir plus sur ce point, vous pouvez vous référer à ce lien)

Ci-dessous, voici la classe `DOM2XPath` en C# :

```
public class DOM2Xpath {
   private XmlDocument doc;

   public DOM2Xpath(String filename) {
       doc = new XmlDocument();
       doc.Load(filename);
   }

   public XmlNodeList getXPath(String nsPrefix, String nsURI, String
       expression) {
       XmlNode root = doc.DocumentElement;
       XmlNamespaceManager nsmgr = new XmlNamespaceManager(doc.NameTable);
       nsmgr.AddNamespace(nsPrefix, nsURI);
       return root.SelectNodes(expression, nsmgr);
```

```
    }

}
```

... et son usage :

```
internal class Program {
    private static void Main(string[] args) {
         // parse le document
         DOM2Xpath bagagesDOM = new DOM2Xpath("./XML/bagagerieHF.xml");
         // récupère la liste identifiants passages ayant des skis
         // la fonction fixe un préfixe (bg) qu'elle associe au namespace de la
             bagagerie
         XmlNodeList nlBagagesDOM = bagagesDOM.getXPath(
                              "bg",
                              "http://www.timc.fr/nicolas.glade/bagages",
                              "//bg:bagage[bg:description='skis']/@idPassager"
                              );
         // itère dans les noeuds pour afficher leur texte (les identifiants)
         foreach(XmlNode node in nlBagagesDOM)
            Console.WriteLine(node.InnerText);
    }
}
```

## 1.5 Avantages et inconvénients de DOM

### Avantages

Le principal avantage de l'utilisation de DOM est que l'arbre intégral du fichier est construit au début de l'application et est mis en cache (en mémoire). L'application peut donc manipuler le contenu du document en parcourant l'arbre à sa guise. Elle peut par conséquent utiliser des expressions *XPath*. Certaines implémentations de DOM (notamment celles que nous utiliserons en Java et javascript) permettent de modifier le contenu d'un document XML (ajout d'éléments, modification du texte, ajout/suppression d'attributs, etc.).

### Inconvénients

Par contre, pour créer un modèle de document DOM, le document doit être lu entièrement et donc stocké en mémoire entièrement, ce qui peut poser des problèmes avec de très gros documents.

L'alternative, c'est les `forward parsers` c'est à dire les analyseurs qui ne sont pas mis en cache et qui ne font que lire en avant.

## 2 Forward parsers

Les forward parsers sont des analyseurs de document qui ne font qu'avancer dans la lecture d'un document. Ils disposent de méthodes pour dire au programme quoi faire lorsqu'ils rencontrent un noeud (un élement, du texte ...). Rien de ce qu'ils rencontrent n'est mis en mémoire. C'est à vous de récupérer les informations rencontrées par le parseur.

Notez que vous pourriez tout à fait décider de récupérer toutes les informations d'un document et les stocker dans un arbre constitué de noeuds de différents types, rangés dans une structure d'arbre, *etc*, ... autrement dit reconstituer un document DOM ! Et pour cause, l'API DOM est ainsi construite : elle utilise un parser en lecture seule pour construire l'instance de document en mémoire dans un arbre. Celui-ci est éditable.

Dans les sections qui suivent nous allons décrire 3 forward parsers, un en C# (`XmlReader`) et 2 en Java (`SAX` et `StAX`).

# 3 *Simple Api for Xml*: SAX [Java]

## 3.1 Principe

SAX (*Simple API for XML*) définit des évènements de lecture d'un document XML qui devront être gérés par une application. 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, etc. Nous verrons que 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.

Le parseur SAX va parcourir le document XML. Lorsqu'il rencontre un marqueur spécifique du document, il génère un événement. Un marqueur rencontré génère un événement. Les principaux événements sont

- Début du document
- Fin du document
- Balise ouvrante
- Balise fermante
- Données texte entre les balises (nœud texte).

Chaque type d'événement est associé à un ensemble d'informations (nom de l'élément, attributs, namespace, caractères, etc.)

SAX est de type *push* streaming. Cela signifie que l'analyseur envoie (pousse) les données XML vers un programme client dès qu'elles arrivent. L'application utilisant SAX ne contrôle pas quels événements sont générés, ni quand ils sont générés.

Pour recevoir les événements déclanchés par SAX, l'application va devoir utiliser le mécanisme de *callback*. C'est-à-dire qu'elle devra implémenter des méthodes (dont la signature est spécifiée dans l'interface `Content Handler`) qui seront appelée par SAX lorsqu'un certain événement intervient.

La notion de *callback* est très utilisée pour les IHM car elle associe des événements d'interaction avec des méthodes.

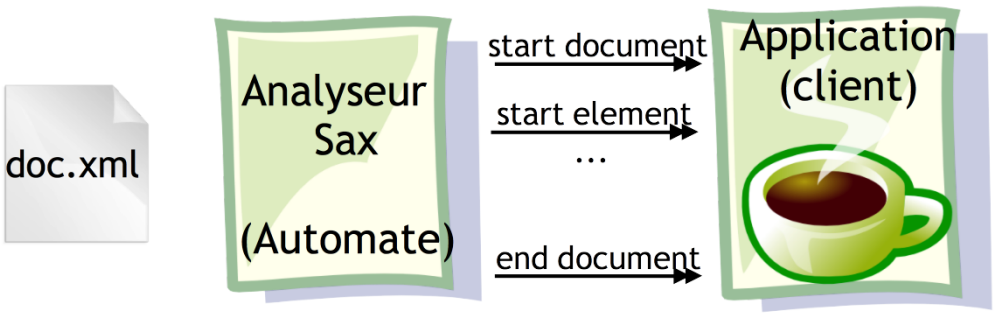


Figure XIII .8: Automate de programmation SAX.

Le parseur produit des événements, gère l'automate de lecture et contrôle le flux. L'application cliente consomme les événements, récupère les données pour les gérer sous une forme liée au domaine d'application, comme illustré figure XIII .8.

La figure XIII .9 présente les différentes parties de SAX

`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, etc.). Dans ce cours, nous nous intéresserons uniquement à l'interface `Content Handler` qui permet de naviguer dans les fichiers. Si vous souhaitez aller plus loin, vous pouvez vous reporter au livre d'où est extrait la figure.

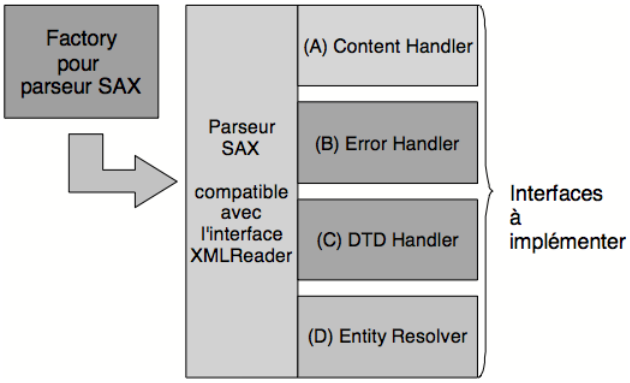


Figure XIII .9: Les différentes parties de SAX. Figure extraite du livre *"XML Cours et Exercies", Alexandre Brillant, Eyrolles Editions.*

## 3.2 Principales méthodes utilisées

Les principaux *Callbacks* sont:

- `startDocument(): void` Début du document
- `startElement(uri: String, localName: String, qName: STring, atts: Attributes): void` Balise ouvrante d'un élément.
- `characters(ch: char[], start: int, length: int): void` Réception d'un texte (arrivée à la fin du texte d'une balise)
- `endElement(uri: String, localName: String, qName: String): void` Arrivée à la fin d'un élément: balise fermante.
- `endDocument(): void` Fin du document

## 3.3 Anatomie d'un application utilisant SAX

Le programme client reçoit des événements (il ne sait pas lequel sera le prochain) et doit donc maintenir un état courant de lecture accessible et modifiable dans toutes les méthodes *callbacks* (par exemples, quelles sont les informations déjà lues, un booléen qui indique l'état (dans une balise, dans du texte, etc.)).

L'application définit généralement une classe qui implémente les méthodes qui seront appelées par le parseur (c'est-à-dire les méthodes définies dans l'interface `Content Handler`), comme illustré figure XIII .10.

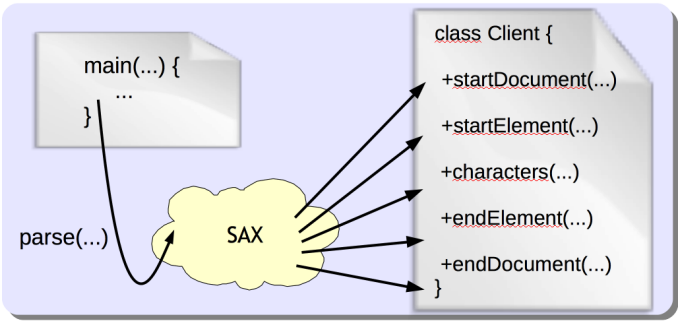


Figure XIII .10: L'application implémente généralement les méthodes définies dans l'interface `Content Handler`.

L'application cliente de SAX (celle que vous allez écrire), doit implémenter dans ces méthodes le code qui correspond à l'action à faire dans chacun des cas. Par exemple, dans `startElement(uri: String, localName: String, qName: STring, atts: Attributes): void`, il peut tester le nom de la balise et avoir des traitements spécifiques en fonction des balises qu'il recherche.

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

/// classe qui implémente les méthodes de l'interface SAX
class MyHandler extends DefaultHandler {
        // attributs qui permettent de stocker les données importantes pour l'
            application
    private int data;
    ...

    @Override
    public void startDocument() throws SAXException {
        // instructions à exécuter au début
    }

    @Override
    public void endDocument() throws SAXException {
        // instructions à exécuter à la fin du document
    }

    @Override
    public void startElement(String uri, String localName, String qName,
        Attributes attributes) throws SAXException     {
                // instructions à exécuter lorsque l'on rencontre une balise
                    ouvrante
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws
        SAXException {
        // instructions à exécuter lorsque l'on rencontre une balise fermante
    }

    @Override
    public void characters(char ch[], int start, int length) throws
        SAXException {
        // instructions à exécuter lorsque l'on rencontre du texte à l'inté
            rieur des balises
    }
}
```

Figure XIII .11: Anatomie d'une application Java utlisant *SAX*.

## 3.4 Exemple

On considère le document XML de la figure XIII .12:

La classe `CompteElements` de la Figure XIII .13 qui implémente les méthodes SAX de l'interface `ContentHandler`.

Enfin, la méthode `main` est représentée figure XIII .14. Dessiner le diagramme APO en notant les évènements pour en déduire ce qui est afficher à l'écran à l'issu de la méthode `main`. La réponse est donnée dans le pdf sur Chamilo.

## 3.5 Avantages et inconvénients

### Avantages

SAX est rapide et ne nécessite que très peu de mémoire. Il est donc très efficace pour les très gros documents ou de nombreux petit documents, et par conséquent, beaucoup utilisé dans les applications web.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<livre lang="fr">
  <titre>XML</titre>
  <auteur>Alexandre Brillant</auteur>
</livre>
```

Figure XIII .12: Document XML à parcourir

```java
import org.xml.sax.*;
public class CompteElement implements ContentHandler {
        private int number;

        public void startDocument() {
                number = 0;
         }
        public void endDocument() {
                System.out.println(number)
        }
        public void startElement(String n, String l, String q, Attributes a) {
                number = number +1;
         }

        public void endElement(String n, String l, String q) { // rein }
        public void characters(char[] text, int start, int length) { // rien }
        public void ignorableWhitespace(char[] text, int start, int length) {
            // rien}

        // Autres méthodes vides...
        // En effet, une implémentation de l'interface ContentHandler se doit d
            'implémenter toutes les méthodes de l'interface pour pouvoir être
            instanciable...
}
```

Figure XIII .13: Classe

```java
public class Main {
        public static void main(String[] argv) {
                XMLReader parser;
                parser = XMLReaderFactory.createXMLReader();
                parser.setContentHandler(new CompteElement());
                parser.parse("unLivre.xml");
        }
}
```

Figure XIII .14: Méthode `main`.

### Inconvénients

SAX ne fournit qu'une analyse séquentielle du document. Il ne permet pas la modification directe du document (en effet, le document n'est pas en mémoire, il n'y a pas d'ajout/modification de contenu possible). Par ailleurs, les requêtes XPath et XQuery possibles en DOM sont impossibles en SAX (ni en StAX d'ailleurs) puisque nécessitent une représentation globale en mémoire.

# 4 *STreaming Api for Xml*: StAX [Java]

## 4.1 Principe

Comme SAX, StAX ne construit pas d'arbre DOM d'un document. Cependant, au lieu d'envoyer des événements à l'application comme SAX, c'est l'application cliente qui pilote la lecture avec StAX. En effet, l'application cliente fait *défiler* le flux de lecture volontairement. StAX utilise donc le *pull streaming*. C'est l'application qui va déclencher la génération de l'avancement dans le flux.

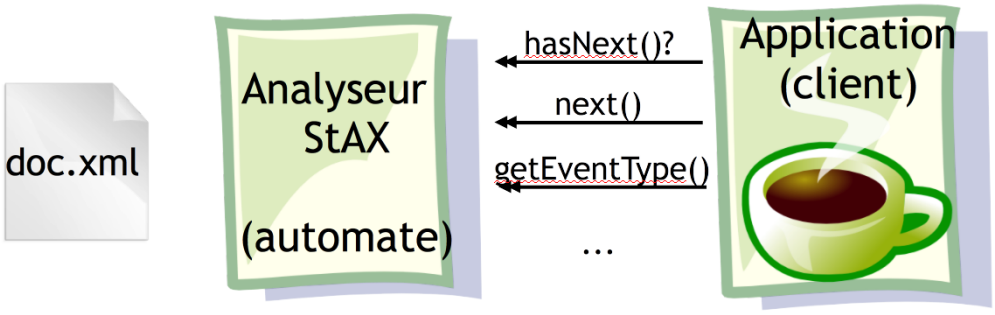


Figure XIII .15: Automate de programmation SAX.

Le *parser* attend les ordres de l'application pour faire avancer l'automate de lecture comme illustré figure XIII .15. L'application cliente envoie les ordres de navigation (essentiellement "aller au prochain événement" ) et traite les informations.

## 4.2 Principales méthodes utilisées

Les principales méthodes d'un *parser* StAX sont

- `hasNext(): boolean` qui indique si le *parser* est arrivé au bout du fichier.
- `next(): int` qui fait avancer le *parser* jusqu'à évènement suivant et renvoie le numéro de l'évènement.
- Les principaux événements possibles sont très proches des événements SAX:
  - `START_DOCUMENT` début du document
  - `START_ELEMENT` balise ouvrante
  - `ATTRIBUTE` un attriut
  - `NAMESPACE` la déclaration d'un espace de nom
  - `CHARACTERS` texte entre les balises
  - `END_ELEMENT` balise fermante
  - `END_DOCUMENT` fin du document
- `getLocalName(): String` renvoie le nom de l'élément
- `getText(): String`
- `getAttributeCount(): int`
- `getAttributeName(index: int): String`
- `getAttributeValue(index: int): String`
- `getAttributeValue(namespaceURI: String, name: String): String`
- `getAttributeName(index: int): String`
- `getAttributeValue(index: int): String`
- `getAttributeValue(namespaceURI: String, name: String): String`

## 4.3 Anatomie d'un application utilisant StAX

Une application cliente de StAX devra instancier un *parser* StAX et piloter la lecture du flux. Pour cela, elle appliquera la méthode `next()` sur le *parser* et récupérera l'entier correspondant à l'événement courant. A chaque événement (étape) de lecture, un aiguillage multiple `switch` permet de faire le traitement adéquat selon le type d'événement courant. La figure XIII .16 illustre ce procédé.

```
1  import javax.xml.stream.XMLInputFactory;
2  import javax.xml.stream.XMLStreamReader;
3
4  import java.io.FileInputStream;
5  public class Test {
6      public static void main(String[] args) {
7          // instanciation du StAX reader
8          XMLInputFactory inputFactory = XMLInputFactory.newInstance();
9          // ici le document en entrée est un fichier, on le lit sous forme de
               stream
10         XMLStreamReader reader = inputFactory.createXMLStreamReader(new
               FileInputStream("nomFichier.xml"),"UTF-8");
11
12         // lecture du fichier tant qu'il y a encore des événèments (i.e des
               choses à lire)
13         while(reader.hasNext()) {
14             // récupérer le prochain (pull)
15             int event = reader.next();
16
17             switch(event) {
18                 case XMLStreamConstants.START_DOCUMENT:
19                     // instructions à exécuter au début du document
20                     break;
21                 case XMLStreamConstants.END_DOCUMENT:
22                     // instructions à exécuter à la fin du document
23                     break;
24                 case XMLStreamConstants.START_ELEMENT:
25                     String elementName: reader.getLocalName();
26                     int nbAttributes = reader.getAttributeCount();
27                     // instructions à exécuter lorsque l'on rencontre une
                           balise ouvrante
28                     break;
29                 case XMLStreamConstants.END_ELEMENT:
30                     // instructions à exécuter lorsque l'on rencontre une
                           balise fermante
31                     break;
32                 case XMLStreamConstants.CHARACTERS:
33                     String text = reader.getText();
34                     // instructions à exécuter lorsque l'on rencontre du texte
                           à l'intérieur des balises
35                     break;
36             }
37         }
38     }
39 }
```

Figure XIII .16: Anatomie d'une application cliente de StAX.

## 4.4 Exemple

On considère le document XML de la figure XIII .17:

La méthode `main` de l'application cliente est donnée dans la figure XIII .18.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<livre lang="fr">
  <titre>XML</titre>
  <auteur>Chagnon</auteur>
</livre>
```

Figure XIII .17: Document XML à parcourir

```java
public class Main {
        public static void main(String[] argv) {
        // Variable qui stocke le nombre d'éléments
        int number;

       // instanciation du StAX reader
        XMLInputFactory inputFactory = XMLInputFactory.newInstance();
        // ici le document en entrée est un fichier, on le lit sous forme de
            stream
        XMLStreamReader reader = inputFactory.createXMLStreamReader(new
            FileInputStream("nomFichier.xml"),"UTF-8");

        // lecture du fichier tant qu'il y a encore des événèments (i.e des
            choses à lire)
        while(reader.hasNext()) {
            // récupérer le prochain (pull)
            int event = reader.next();

            switch(event) {
                case XMLStreamConstants.START_DOCUMENT:
                    number = 0;
                    break;
                case XMLStreamConstants.END_DOCUMENT:
                    System.out.println(number);

                    break;
                case XMLStreamConstants.START_ELEMENT:
                    number += 1;
                    break;
                default:
                    // Dans tous les autres cas, on ne fait rien
                    break;
            }
        }
    }
}
```

Figure XIII .18: Méthode `main`.

## 4.5 Avantages et inconvénients

Les bibliothèques *pull* sont plus légères que les bibliothèques *push* ou DOM. Elle sont donc souvent utilisées dans les smartphones, tablettes, etc. De plus, comme c'est l'application cliente qui contrôle la lecture, elle peut ainsi *sauter* des données qui lui sont inutiles.

# 5 La classe XmlReader [C#]

## 5.1 Principe

La classe XmlReader de C# est proche de StAX de Java. Elle permet une navigation en avant contrôlée par l'application (pull). Cette classe dispose en plus d'autres méthodes permettant notamment la validation de documents XML par rapport à un Schema XML ou de vérifier leur conformité.

Comme pour StAX c'est l'application cliente qui pilote la lecture : elle fait défiler le flux de lecture volontairement sous le controle de la méthode `Read() : bool` qui lit le noeud suivant et fait avancer le flux de lecture.

Notez que `XmlReader` supporte la programmation asynchrone. Les méthodes sont déclinées en version asynchrone (leur nom comporte `Async` à la fin).

## 5.2 Principales méthodes et propriétés utilisées

Les principales méthodes et propriétés d'un *XmlParser* sont:

- `Create(...)` une méthode pour créer un lecteur à partir d'un nom de fichier, d'un flux (`Stream`) ou d'un lecteur de texte `TextReader`.
- `Read(): bool` qui lit le noeud suivant si le *parser* n'est pas arrivé au bout du fichier, récupère ses informations et les stocke dans l'instance d'`XmlReader`. L'ensemble de ce qui est récupéré à chaque avancement d'un noeud est accessible via des propriétés, par exemple le type de noeud (propriété `NodeType` de type énuméré `XmlNodeType`).
- Les propriétés principales :
  - `NodeType : XmlNodeType`. Les principaux `XmlNodeType` possibles sont : `Attribute`, `Comment`, `Document`, `Element` (qui indique le début d'un élément), `EndElement`, `Text`, `Whitespace` (un espace avant un chevron).
  - `IsEmptyElement : bool` renvoie false si l'élément est vide, par exemple `<root/>`
  - `HasAttribute : bool` est ce que l'élement (si l'on est dans un élement) contient un ou plusieurs attributs ?
  - `LocalName` le nom de l'élement (si l'on est dans un élement)
  - `Value` la valeur contenue dans l'élement s'il s'agit de texte, de commentaires, de CData.
- Les méthodes de type "MoveTo...". La classe `XmlReader` comporte plusieurs méthodes dont le nom comporte `MoveTo`. Ces mouvements peuvent se faire localement en arrière (dans un élément). Voici ces méthodes :
  - `MoveToContent() : XmlNodeType` Cette méthode fait avancer le curseur de lecteure jusqu'au premier noeud dit "de contenu", autrement dit tout ce qui n'est pas commentaires, balises `DocumentType`, espaces ...
  - `MoveToAttribute(Int32)` et `MoveToAttribute(String)` permettent de se déplacer vers un attribut donné (par index ou par nom).
  - `MoveToFirstAttribute()` et `MoveToLastAttribute()` qui déplacent le curseur respectivement vers le premier et le dernier attribut.
  - `MoveToElement()` : cette méthode est particulière puisque c'est la seule, dans une certaine mesure, capable de remonter un peu le flux de lecture, localement seulement. Elle permet de remonter au niveau de l'élément contenant l'attribut constituant le noeud auquel on se trouve. On peut donc explorer les attributs d'un éléments puis remonter vers leur élément.
- `Skip()` permet d'ignorer les enfants de l'élément actuel.
- Les méthodes `IsStartElement():bool` et `IsStartElement(name: String):bool` : ces méthodes appellent `MoveToContent()` et vérifie si le noeud actuel est une balise de début (début d'élément) ou une balise d'élement vide. La deuxième version permet de vérifier en plus si l'élément courant dans lequel on vient d'entrer a pour nom la valeur `name` passée en paramètre.

Il y a bien d'autres méthodes dans cette classe mais avec celles-ci, on peut déjà lire un document.

## 5.3 Anatomie d'un application utilisant XmlReader

Une application cliente de XmlReader devra instancier un *parser* XmlReader et piloter la lecture du flux. Pour cela, elle appliquera la méthode `Read()` sur le *parser* et récupérera le `XmlNodeType` correspondant à l'événement courant (noeud rencontré). A chaque étape de lecture, un aiguillage multiple `switch` permet de faire le traitement adéquat selon le type d'événement courant. Le listing ci-dessous illustre ce procédé implémenté dans une classe C# nommée `XMLUtils`.

```
1  using System.Xml;
2  using System.Xml.Schema;
3
4  public static class XMLUtils {
5
6      public static void Analyze(string filepath) {
7          var settings = new XmlReaderSettings();
8          using (var reader = XmlReader.Create(filepath, settings)) {
9              reader.MoveToContent();
10             while (reader.Read())
11                 switch (reader.NodeType) {
12                     case XmlNodeType.XmlDeclaration:
13                         // instructions à executer quand le prompt est détecté
14                         Console.Write("Found XML declaration (<?xml version
                               ='1.0'?>)");
15                         break;
16                     case XmlNodeType.Document:
17                         // instructions à exécuter au début du document
18                         Console.Write("Entering the document");
19                         break;
20                     case XmlNodeType.Comment:
21                         // instructions à executer quand on trouve un
                               commentaire
22                         Console.Write("Comment = <!--{0}-->", reader.Value);
23                         break;
24                     case XmlNodeType.Element:
25                         // instructions à executer quand on entre dans un élé
                               ment
26                         Console.WriteLine("Starts the element {0}", reader.Name
                               );
27                         break;
28                     case XmlNodeType.Attribute:
29                         // instructions à executer quand on trouve un attribut
30                         Console.WriteLine("Attribute {0}", reader.Name);
31                         break;
32                     case XmlNodeType.CDATA:
33                         // instructions à executer quand on trouve un bloc
                               CData
34                         Console.Write("Found CData part : <![CDATA[{0}]]>",
                               reader.Value);
35                         break;
36                     case XmlNodeType.Text:
37                         // instructions à executer quand on trouve du texte
38                         Console.WriteLine("Text node value = {0}",
39                             reader.GetValueAsync());
40                         break;
41                     case XmlNodeType.EndElement:
42                         // instructions à executer quand on sort d'un élément
43                         Console.WriteLine("Ends the element {0}", reader.Name);
44                         break;
45                     default:
46                         // instructions à executer sinon
47                         Console.WriteLine("Other node of type {0} with value
                               {1}",
48                             reader.NodeType, reader.Value);
49                         break;
```

```
                }
            }
        }

}
```

... et sa version asynchrone :

```
    public static async Task AnalyzeAsync(string filepath) {
        var settings = new XmlReaderSettings();
        settings.Async = true;
        using (var reader = XmlReader.Create(filepath, settings)) {
            await reader.MoveToContentAsync();
            while (await reader.ReadAsync())
                switch (reader.NodeType) {
                    case XmlNodeType.XmlDeclaration:
                        Console.Write("Found XML declaration (<?xml version
                            ='1.0'?>)");
                        break;
                    case XmlNodeType.Comment:
                        Console.Write("Comment = <!--{0}-->", reader.Value);
                        break;
                    case XmlNodeType.Document:
                        break;
                    case XmlNodeType.Element:
                        Console.WriteLine("Starts the element {0}", reader.Name
                            );
                        break;
                    case XmlNodeType.Attribute:
                        // instructions à executer quand on trouve un attribut
                        Console.WriteLine("Attribute {0}", reader.Name);
                        break;
                    case XmlNodeType.CDATA:
                        Console.Write("Found CData part : <![CDATA[{0}]]>",
                            reader.Value);
                        break;
                    case XmlNodeType.Text:
                        Console.WriteLine("Text node value = {0}",
                             await reader.GetValueAsync());
                        break;
                    case XmlNodeType.EndElement:
                        Console.WriteLine("Ends the element {0}", reader.Name);
                        break;
                    case XmlNodeType.Whitespace: // if removed -> Other node
                        below
                        Console.WriteLine("[WS]");
                        break;
                    default:
                        Console.WriteLine("Other node of type {0} with value
                            {1}",
                             reader.NodeType, reader.Value);
                        break;
                }
        }
    }
```

## 5.4 Exemple

Voici un exemple de parser écrit avec `XmlReader`. La classe statique `Bagagerie` deux méthodes qui parsent différemment une instance de document de bagagerie à partir d'un nom de fichier, les méthodes statiques `CountBagages(filename: String):int` qui compte le nombre de bagages et `GetIDPassagers(filename: String): List<String>` qui récupère dans une liste de chaînes de caractères les identifiants des pas-

sagers. Remarquez que cette dernière utilise un ensemble `HashSet` pour récupérer les valeurs des identifiants, évitant ainsi les doublons.

```
public static class BagagerieXmlReader {

    static public int CountBagages(String filename) {
        XmlReader reader = XmlReader.Create(filename);
        int nbBagages = 0;
            List<String> ids = new List<string>();
            reader.MoveToContent();
            while (reader.Read())
                switch (reader.NodeType) {
                    case XmlNodeType.Element:
                        // instructions à executer quand on entre dans 1 élément
                        if (reader.IsStartElement("bagage"))
                            nbBagages++;
                        break;
                }
            return nbBagages;
    }

    static public HashSet<String> GetIDPassagers(String filename) {
        XmlReader reader = XmlReader.Create(filename);
        HashSet<string> ids = new HashSet<string>();
        while (reader.Read())
            switch (reader.NodeType) {
                case XmlNodeType.Element:
                    if (reader.Name == "bagagerie")
                        Console.WriteLine("Commence à explorer la bagagerie");
                    if (reader.IsStartElement("bagage")) {
                        reader.MoveToFirstAttribute();
                        Console.WriteLine("   -> un attribut {0} trouvé, de valeur {1}",reader.Name, reader.Value);
                        ids.Add(reader.Value);
                    }
                    break;
                case XmlNodeType.EndElement:
                    if (reader.Name == "bagagerie") {
                        Console.WriteLine("... et voila c'est fini");
                    }
                    break;
            }
        return ids;
    }
}
```

et son usage dans la fonction Main ...

```
Console.WriteLine("Nombre de bagages comptés : "+BagagerieXmlReader.CountBagages("./XML/bagagerieHF.xml"));
HashSet<string> ids = BagagerieXmlReader.GetIDPassagers("./XML/bagagerieHF.xml");
Console.WriteLine("Liste des identifiants passagers récupérés :");
foreach (var id in ids)
    Console.WriteLine(" -> "+id);
```

donne en sortie :

```
Nombre de bagages comptés : 4
Commence à explorer la bagagerie
   -> un attribut idPassager trouvé, de valeur OK17401023
   -> un attribut idPassager trouvé, de valeur OK17401023
   -> un attribut idPassager trouvé, de valeur AG18300427
   -> un attribut idPassager trouvé, de valeur SF23890311
```

```
... et voila c'est fini
Liste des identifiants passagers récupérés :
 -> OK17401023
 -> AG18300427
 -> SF23890311
```

# 6 Quelques usages des parsers en C#

## 6.1 Validation de schémas avec .NET XmlReader

Il est possible avec `XmlReader` de valider un document XML avec le Schema qui le modélise. Pour cela, il suffit de configurer l'instance de `XmlReaderSettings` qui est fournie au reader. Dans les settings on peut fournir notamment des schémas XML grâce à la méthode Add(namespaceURI, filename) de la propriété `Schemas` des settings. On spécifie le type de validation (ici `ValidationType.Schema`). On ajoute également une méthode extérieure (un `Callback` (cf. cours de prog fonctionnelle)) qui sera appelée si une alerte (warning) ou une erreur est détectée lors de la validation. On parse alors le document en appelant la méthode `Read(): bool` jusqu'à ce que la fin du fichier soit atteinte. Vous trouverez ci-dessous le code complet permettant de valider un document (fichier XML) avec un schéma (fichier XSD).

```
public static async Task ValidateXmlFile(string schemaNamespace, string
    xsdFilePath, string xmlFilePath) {
        var settings = new XmlReaderSettings();
        settings.Schemas.Add(schemaNamespace, xsdFilePath);
        settings.ValidationType = ValidationType.Schema;
        Console.WriteLine("Nombre de schemas utilisés dans la validation : "+
            settings.Schemas.Count);
        settings.ValidationEventHandler += ValidationCallBack;
        var readItems = XmlReader.Create(xmlFilePath, settings);
        while (readItems.Read()) { }
    }


    private static void ValidationCallBack(object? sender, ValidationEventArgs
        e) {
        if (e.Severity.Equals(XmlSeverityType.Warning)) {
            Console.Write("WARNING: ");
            Console.WriteLine(e.Message);
        }
        else if (e.Severity.Equals(XmlSeverityType.Error)) {
            Console.Write("ERROR: ");
            Console.WriteLine(e.Message);
        }
    }
```

et le code d'appel :

```
XMLUtils.ValidateXmlFile("http://www.timc.fr/nicolas.glade/bagages", "./XML/
    bagagerieHF.xml", "./XML/bagagerieHF.xsd");
```

## 6.2 Génération de schémas avec .NET XmlReader et XmlSchema

`XmlReader` permet en outre de générer un schéma XML à partir d'un document XML ! Alors, attention, je mets tout de suite en garde les étudiants qui seraient très tentés de procéder ainsi pour éviter de se creuser la tête à écrire un beau schéma. Les schémas XML qui sont générés sont certes tout à fait corrects, mais absolument pas modulaires : ils utilisent des types anonymes, encastrés les uns dans les autres. Ca marche, mais c'est moche et peu réutilisable : on n'a plus qu'un seul type. De plus, les restrictions que vous auriez souhaitées n'y sont pas ; forcément, elles n'y sont pas non plus dans votre code C#. De même pour les contraintes d'existence ou d'unicité. La bonne nouvelle pour moi ceci dit,

c'est que si vous avez lu ce message, c'est que visiblement vous lisez attentivement le cours. C'est très bien, continuez !

Pour faire cela, il faut se servir de la classe `XmlSchemaInference`. Un tel objet, à qui l'on peut fournir la connaissance d'un document XML via le XmlReader qui le parse, possède une méthode `InferSchema(reader: XmlReader) : XmlSchemaSet` qui renvoie un ensemble de schémas modélisant ce que lui indique le reader.

Voici le code qui fait cela :

```
public static string GenerateXSD(string xmlFilePath) {
        var reader = XmlReader.Create(xmlFilePath);
        var schema = new XmlSchemaInference();
        var schemaSet = schema.InferSchema(reader);
        var xsdText = "";
        foreach (XmlSchema s in schemaSet.Schemas())
            using (var stringWriter = new StringWriter()) {
                using (var writer = XmlWriter.Create(stringWriter)) {
                    s.Write(writer);
                }
                xsdText += stringWriter.ToString();
            }
        return xsdText;
    }
```

## 6.3 Transformations XSLT depuis le langage appelant [C#]

En C#, appliquer une tranformation XSLT est simple : il faut charger l'instance de document XML dans un `XPathDocument`, puis instancier une transformation de type `XslCompiledTransform` à partir du fichier XSLT, puis appliquer la transformation, comme le montre le listing suivant.

```
public void XslTransform(string xmlFilePath, string xsltFilePath, string
    htmlFilePath) {
        XPathDocument xpathDoc = new XPathDocument(xmlFilePath) ;
        XslCompiledTransform xslt = new XslCompiledTransform();
        xslt.Load(xsltFilePath);
        XmlTextWriter htmlWriter = new XmlTextWriter(htmlFilePath,null);
        xslt.Transform(xpathDoc,null,htmlWriter);
    }
```

# 7 Résumé des différents *parsers*

| Propriété | DOM | SAX (Java) | StAX (Java) | XmlReader (C#) |
|---|---|---|---|---|
| Type d'API | Arborescence | *Push streaming* | *Pull streaming* | *Pull streaming* |
| Utilisation XPath | Oui | Non | Non | Non |
| Efficacité (CPU & mémoire) | Variable | Excellente | Excellente | Excellente |
| Lecture de document XML | Oui | Oui | Oui | Oui |
| Retour arrière dans un document XML | Oui | Non | Non | Non (navigation attribut |
| Écriture de document XML | Oui | Non | Non | Non |
| Création/Modification de nœud | Oui | Non | Non | Non |
| Types d'applications | - Modification de Documents XML<br>- Utilisation XPath, XSLT, etc.<br>- Accès aléatoire dans un arbre XML<br>- Fusion de documents | - Lecture de documents<br>- Efficacité de la mémoire<br>- Systèmes embarqués<br>- Recherche de certains nœuds | - Lecture de documents<br>- Lecture simultanée de plusieurs documents<br>- Efficacité de la mémoire<br>- Systèmes embarqués<br>- Recherche de certains nœuds | - Lecture de documents<br>- Lecture simultanée de plusieurs documents<br>- Lecture asynchrone<br>- Efficacité de la mémoire<br>- Systèmes embarqués<br>- Recherche de certains nœuds |