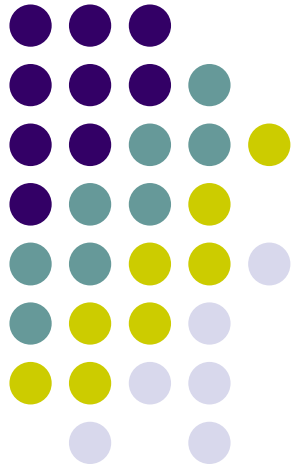
Interfaces de Programmation pour XML

DOM & SAX

Pr. Sidi Mohammed Benslimane
École Supérieure en Informatique
08 Mai 1945 – Sidi Bel Abbes –

s.benslimane@esi-sba.dz



Présentation

- Une Interface de Programmation (*Application Programming Interface* API en anglais) est un ensemble de bibliothèques de fonctions et d'outils permettant d'écrire des programmes spécialisés.
- Les APIS XML permettent et facilitent la lecture, la génération et le traitement (recherche, transformation, validation, sérialisation, etc.) de documents XML.
- Deux approches permettent aux programmeurs de manipuler des documents XML:
  - Utilisation de la forme arborescente XML: DOM
  - Utilisation de la forme sérialisée XML: SAX

Modèles d'implémentation

- L'API DOM (*Document Object Model*) manipule une représentation d'un document complet. La totalité du document est chargée en mémoire pendant le traitement.
- ***Modèle Hiérarchique*** : DOM XML permet la manipulation d'un document XML après l'avoir représenté en mémoire sous la forme d'un arbre d'objets.
  - Avantage: utilisation simple, la plus utilisée, permet la lecture et l'écriture.
  - Inconvénient: stockage en mémoire de la structure d'arbre, difficulté et lourdeur de traiter de gros fichiers XML.

Modèles d'implémentation

- L'API SAX (*Simple API for XML*) permet de lire un document XML de manière séquentielle sans rien stocker en mémoire.
- ***Modèle Événementiel*** : dans SAX, le traitement des données est basé sur une gestion d'événements (début/fin d'un document, début/fin d'un élément, etc.).
  - Avantage: pas besoin de stocker le document en mémoire, rapidité,
  - Inconvénient: programmation complexe, ne permet pas l'écriture.

Modèles d'implémentation

- DOM est une norme du consortium w3

http://www.w3.org/DOM/

- SAX est une spécification disponible en projet Open Source.

http://www.saxproject.org/

- Il existe de nombreuses implémentations de la norme DOM et de la spécification SAX dans plusieurs langages de programmation:

- Java,
- JavaScript,
- PHP,
- Python,
- C++,

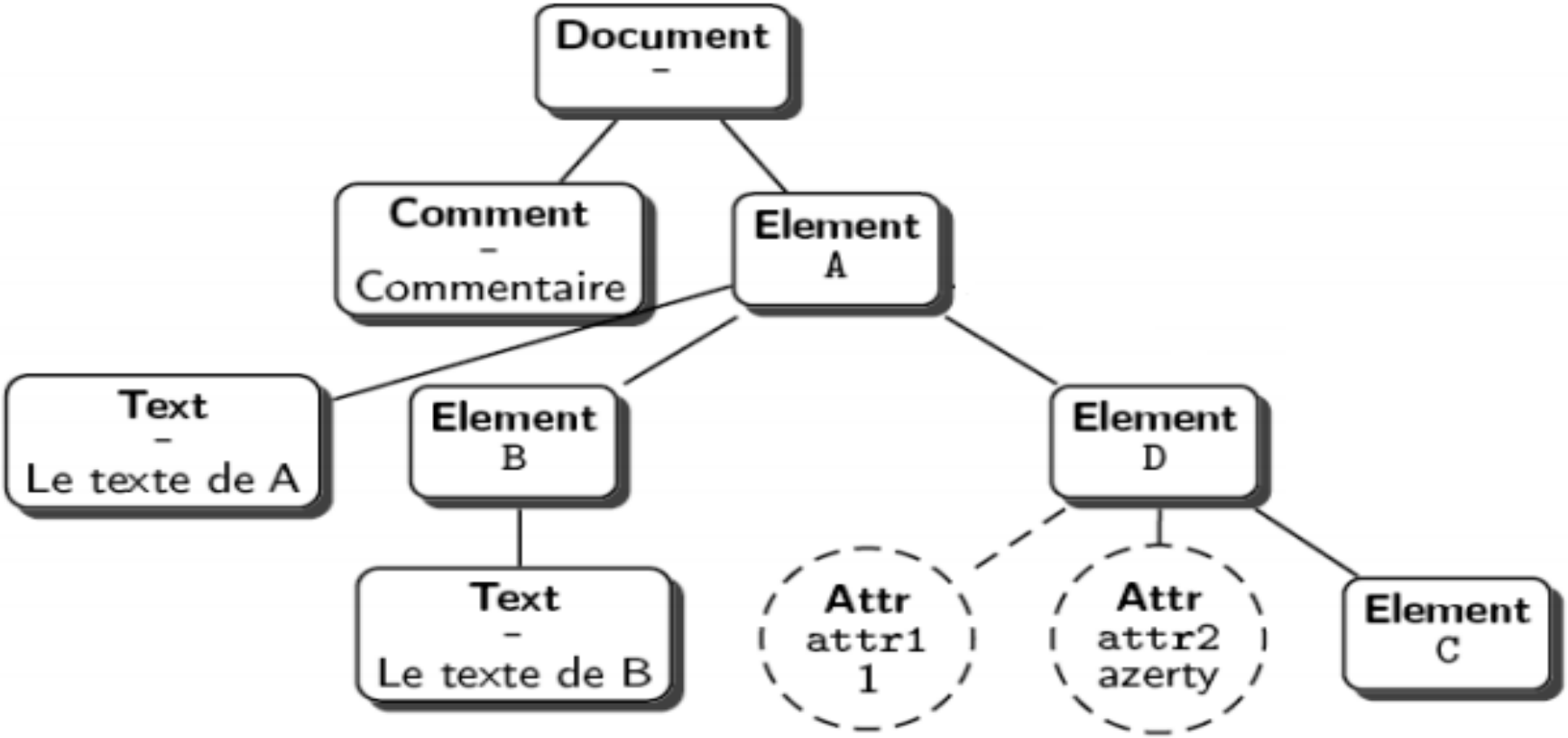
Principes généraux de l'API DOM

- Lors de la création ou de l'ouverture d'un document XML, une instance qui représente le document tout entier est créée.
- Par la suite, les méthodes de cette instance sont utilisées pour créer ou parcourir les éléments, attributs et textes du document. Il existe deux modes de manipulations:

1. En mode création d'un document:
   1. Créer une instance de **Document**,
   2. Ajouter des instances d'**Element** au document, leur ajouter des **attributs**, textes, comentaires, etc,
   3. Écrire le document dans un fichier.
2. En mode lecture d'un document:
   1. Créer une instance de **Document**,
   2. Ouvrir et analyser un fichier XML, ça remplit le document,
   3. Parcourir les instances d'**Element** du document.

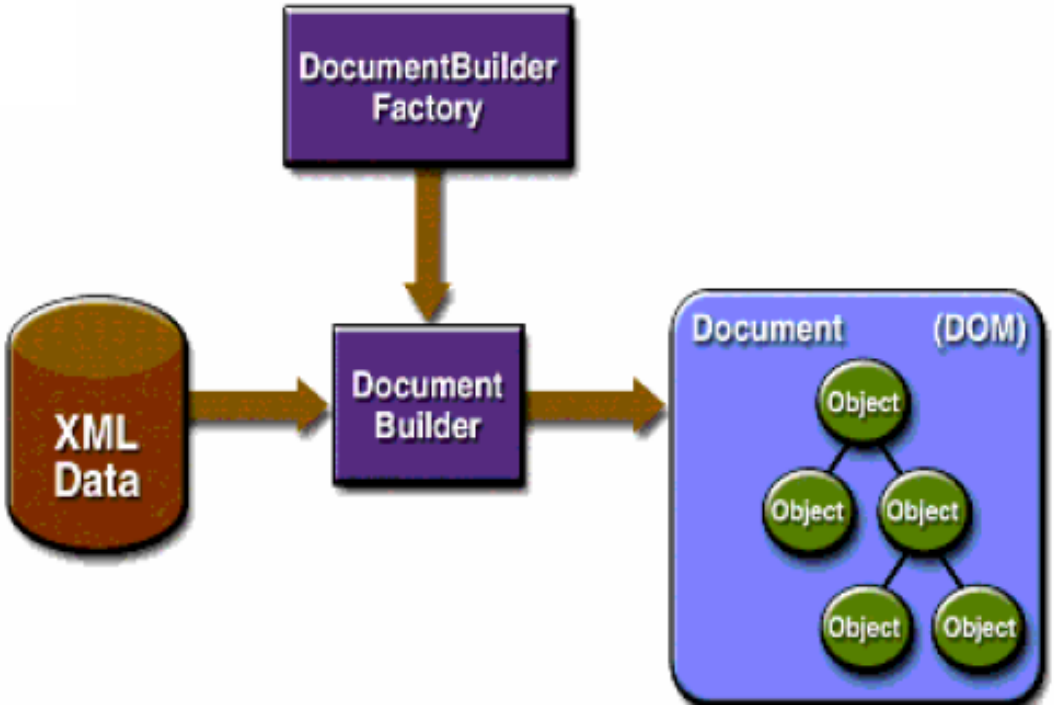
Document DOM en mode création

Représentation DOM



Création d'un Document

1. La Fabrique de parseurs ("DocumentBuilder Factory") produit le Parseur DOM ("Document Builder"),
2. Le Parseur DOM ("Document Builder") transforme un document XML en arbre DOM
3. L'arbre DOM est manipulé à l'aide des méthodes de l'API qui permettent le parcours, la modification, etc.



Création d'un Document

- En Java, il faut trois instructions :

```
// création d'une fabrique de parseurs (Factory)
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();


// création d'un parseur (Builder)
DocumentBuilder builder = factory.newDocumentBuilder();


// création d'un document
Document document = builder.newDocument();
```

Bibliothèques

- Pour travailler avec l'API, il faut importer un certain nombre de librairies :

```
import java.io.File;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;
```

Création d'un Document

- Le code complet se présente comme ceci :

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
void CreationXML()
{
 try {
 DocumentBuilderFactory factory= DocumentBuilderFactory.newInstance();
 DocumentBuilder builder= factory.newDocumentBuilder();
 Document document = builder.newDocument();
  ...}
  catch (Exception e) {...}
}
```

Création d'éléments

- La classe Document possède des méthodes pour rajouter des éléments. Ça se passe en deux temps :

1. Création d'un élément : **`document.createElement(nom);`**
2. Ajout de cet élément dans le document, en tant qu'enfant d'un élément existant : **`parent.appendChild(enfant);`**

```
// création de la racine du document
Element racine = document.createElement("voiture");
document.appendChild(racine);
// ajout d'un élément sous la racine
Element marque = document.createElement("marque");
racine.appendChild(marque);
```

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

Création d'un arbre d'éléments

- On pourrait créer des éléments à la volée de cette manière:

```
// ajout de plusieurs éléments sous la racine
racine.appendChild(document.createElement("marque"));
racine.appendChild(document.createElement("couleur"));
```

- Cependant, nous n'avons aucune variable pour représenter les éléments rajoutés. Nous ne pouvons pas leur rajouter des enfants et des attributs.
- Pour créer un arbre complexe, il faut définir des variables pour chacun des éléments. Cela peut passer par des tableaux :

```
Element annees[] = new Element[4];
for (int i=0; i<4; i++)
   {
     annees[i] = document.createElement("annee");
     racine.appendChild(annees[i]);
    }
```

Ajout d'attributs aux éléments

- Placer des attributs sur un élément est très facile. On peut manipuler l'attribut en tant qu'objet :

```
import org.w3c.dom.Attr;
Attr attribut = document.createAttribute("attribut");
attribut.setValue("valeur");
element.setAttributeNode(attribut);
```

- ou plus simplement :

```
element.setAttribute("attribut", "valeur");
```

- Exemple:

```
element.setAttribute("couleur", "verte");
```

Ajout de texte et commentaires

- C'est simple de rajouter du texte :

```
import org.w3c.dom.Text;
element.appendChild(document.createTextNode("texte"));
```

- C'est aussi simple de rajouter du commentaire:

```
import org.w3c.dom.Comment;
Comment commentaire = document.createComment("commentaire");
element.appendChild(commentaire);
```

Enregistrement dans un fichier

- C'est à faire tout à la fin, lorsque le document est complet.

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

// création de la fabrique
TransformerFactory transformerFactory = TransformerFactory.newInstance();

// récupération du transformeur
Transformer transformer =  transformerFactory.newTransformer();

// configuration du transformeur
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

// écriture du document dans un fichier
DOMSource source = new DOMSource(document);
StreamResult sortie =new StreamResult(new File("sortie.xml"));
transformer.transform(source, sortie);
```

Document DOM en mode lecture

Traitement du document

- On se place maintenant du côté lecture et analyse d'un document XML existant.

- La problématique consiste à :
  - Chercher un ou plusieurs noeuds spécifiques,
  - Itérer sur tous les noeuds enfants d'un noeud,
  - Vérifier le nom d'un noeud,
  - Extraire le contenu texte d'un nœud,
  - Extraire les valeurs d'attributs d'un noeud.
  - etc.

- C'est en général un ensemble de tout cela.

Types de programmation DOM

- **Deux façons de programmer le parcours d'un arbre DOM**

1. Voir tous les nœuds comme des objets de type Node

✓ Avantage: arbre homogène, plus simple pour un parcours complet

✓ Inconvénient: méthodes simples (communes)

2. Voir chaque nœud avec son type spécifique

✓ Avantage: méthodes spécifiques pour chaque type, plus puissantes

✓ Inconvénient: arbres hétérogènes, plus difficile à programmer

- **En pratique: deux méthodes principales**

1. Parcours de l'arbre DOM comme un arbre d'objets Node + actions en fonction du type du nœud courant
2. Parcours des nœuds Element (balises) guidé par leur nom et accès à leurs attributs, contenus textuels, etc.

Ouverture d'un fichier

- Pour ouvrir un fichier XML existant, le début est similaire à la création d'un document :

```
1. DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
2. DocumentBuilder builder = factory.newDocumentBuilder();
3. factory.setNamespaceAware(true);
4. Document document = builder.parse("document.xml");
```

- Notez les deux changements :
  - (ligne 3) On prévient qu'il va y avoir des *namespaces*,
  - (ligne 4) On remplit le document avec ce qui se trouve dans le fichier XML.

L'interface Node

- Représente un noeud de l'arborescence d'un document XML.
- Tous les différents noeuds qui composent l'arbre héritent de l'interface org.w3c.dom.Node.
- L'interface définit plusieurs méthodes :

Méthode	Rôle
short getNodeType()	Renvoyer le type du noeud
String getNodeName()	Renvoyer le nom du noeud
String getNodeValue()	Renvoyer la valeur du noeud
NamedNodeList getAttributes()	Renvoyer la liste des attributs ou null
void setNodeValue(String)	Mettre à jour la valeur du noeud
boolean hasChildNodes()	Renvoyer un booléen qui indique si le noeud a au moins un noeud fils
Node getFirstChild()	Renvoyer le premier noeud fils du noeud ou null
Node getLastChild()	Renvoyer le dernier noeud fils du noeud ou null
NodeList getChildNodes()	Renvoyer une liste des noeuds fils du noeud ou null
Node getParentNode()	Renvoyer le noeud parent du noeud ou null
Node getPreviousSibling()	Renvoyer le noeud frère précédent
Node getNextSibling()	Renvoyer le noeud frère suivant
Node insertBefore(Node, Node)	Insèrer le premier noeud fourni en paramètre avant le second noeud
Node replaceNode(Node, Node)	Remplacer le second noeud fourni en paramètre par le premier
Node removeNode(Node)	Supprimer le noeud fourni en paramètre
Node appendChild(Node)	Ajouter le noeud aux noeuds enfants du noeud courant
Node cloneNode(boolean)	Renvoyer une copie du noeud. Le booléen fourni en paramètre indique si la copie doit inclure les noeuds enfants

L'interface Node

- La méthode getNodeType() permet de connaître le type du noeud. Le type est très important car il permet de savoir ce que contient le noeud.

Constante	Valeur	Rôle	Observation
ELEMENT_NODE	1	Elément	
ATTRIBUTE_NODE	2	Attribut	
TEXT_NODE	3	Texte	
PROCESSING_INSTRUCTION_NODE	7	Instruction de traitement	
COMMENT_NODE	8	Commentaire	
DOCUMENT_NODE	9	Racine du document	
DOCUMENT_TYPE_NODE	10	Document	
DOCUMENT_FRAGMENT_NODE	11	Fragment de document	

L'interface Document

- L'interface Document hérite de l'interface Node.
- Elle définit plusieurs méthodes :

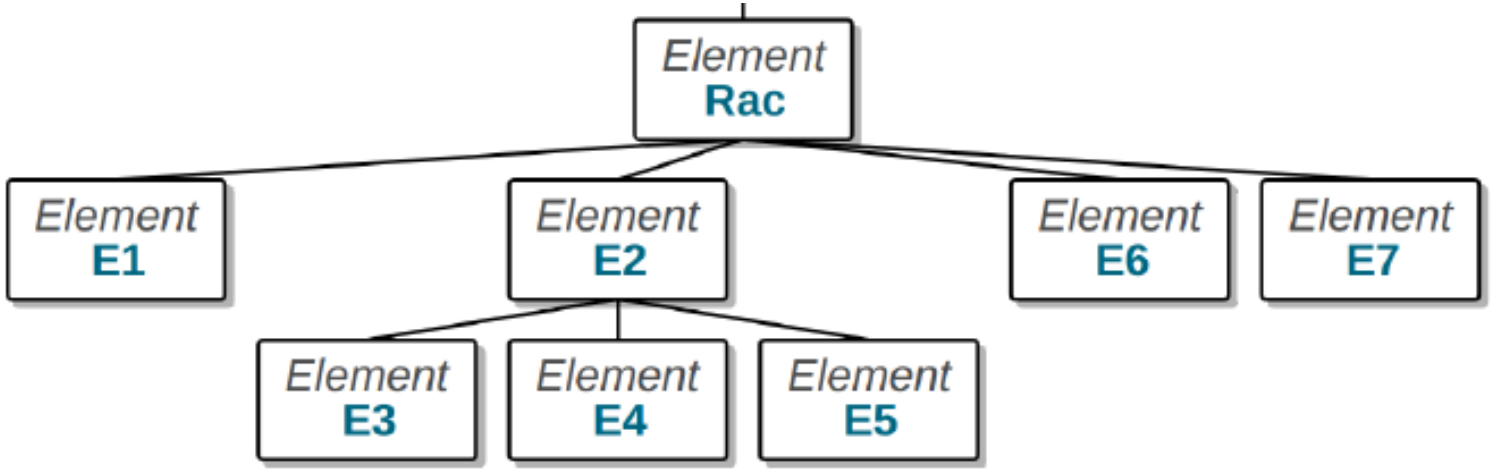
Méthode	Rôle
Element getDocumentElement()	Renvoyer l'élément racine du document
Attr createAttributes(String)	Créer un attribut dont le nom est fourni en paramètre
Comment createComment(String)	Créer un noeud de type commentaire
Element createElement(string)	Créer un noeud de type élément dont le nom est fourni en paramètre
DocumentType getDocType()	Renvoyer les informations sur le type de document (DTD)

L'interface Element

- L'interface Element hérite de l'interface Node
- Elle définit des méthodes pour manipuler un élément et en particulier les attributs d'un élément.
- Un objet de type Element à toujours pour type de noeud ELEMENT_NODE

Méthode	Rôle
NodeList getElementsByTagName(String)	Renvoyer une liste des noeuds enfants dont le nom correspond au paramètre fourni
String getAttribute(String)	Renvoyer la valeur de l'attribut dont le nom est fourni en paramètre
removeAttribut(String)	Supprimer l'attribut dont le nom est fourni en paramètre
setAttribut(String, String)	Modifier ou créer un attribut dont le nom est fourni en premier paramètre et la valeur en second
Attr getAttributeNode(String)	Renvoyer un objet de type Attr qui encapsule l'attribut dont le nom est fourni en paramètre
Attr removeAttributeNode(Attr)	Supprimer l'attribut fourni en paramètre
Attr setAttributNode(Attr)	Modifier ou créer un attribut

Voisinage d'un noeud



Une instance de la classe Element peut avoir un Node parent, des enfants, ainsi que des frères.

`E2.getParentNode()` renvoie le noeud Rac
`E2.getPreviousSibling()` renvoie le noeud frère précédent E1
`E2.getNextSibling()` renvoie le noeud frère suivant E6
`E2.getFirstChild()` renvoie le premier noeud enfant E3
`E2.getLastChild()` renvoie le dernier noeud enfant E5

Toutes ces méthodes retournent null si aucun Node ne correspond.

Modification d'un document

- Les méthodes suivantes permettent de modifier un document :

- *element*.appendChild(*node*) ajoute le noeud (élement, texte, etc.) après tous les enfants de l'élément.
- *element.insertBefore(node,autre)* ajoute le nœud (élement, texte, etc.) avant *autre* parmi les enfants de l'élément.
- *element.removeChild(node)* retire le noeud indiqué de la liste de l'élément.
- *document.renameNode(node, URI, nom_qualifié)* change le nom du noeud indiqué. Mettre URI à null s'il n'y a pas de *namespace*.

  Exemple: `document.renameNode("nom",null,"name");`

Élément racine

- On obtient l'objet Java représentant la racine du document par:

```
Element racine = document.getDocumentElement();
```

- NB: cet élément est unique, sinon le fichier XML est mal formé.
- Pour avoir le nom de la racine, on emploie l'un de ses getters :

```
String nom = racine.getNodeName();
```

Parcours des noeuds enfants (méthode 1)

- Pour passer les enfants d'un élément en revue, on peut utiliser l'algorithme suivant :

```
Node courant = element.getFirstChild();
while (courant != null)
      {
        // traiter le noeud courant
        ...
       // passer au suivant
       courant = courant.getNextSibling();
      }
```

Parcours des noeuds enfants (méthode 2)

- On peut aussi utiliser la méthode getChildNodes() qui retourne une liste de Node dans un objet de type NodeList.
- C'est une sorte de tableau dont on peut récupérer la taille et l'un des éléments par son indice.

```
NodeList liste = element.getChildNodes();
final int nombre = liste.getLength();
for (int i=0; i<nombre; i++)
      {
       Node courant = liste.item(i);
       // traiter le noeud courant
       ...
      }
```

Parcours des noeuds enfants (méthode 3)

- Il y a encore une autre manière de parcourir certains enfants d'un élément, en utilisant la méthode getElementsByTagName(nom) qui retourne une NodeList des éléments ayant le nom indiqué.

```
NodeList liste = element.getElementsByTagName("voiture");
final int nombre = liste.getLength();
for (int i=0; i<nombre; i++)
     {
       Node courant = liste.item(i);
       // traiter le noeud courant
       ...
      }
```

Parcours des noeuds enfants (méthode 4)

- Il existe enfin une quatrième manière pour trouver directement les éléments qu'on souhaite dans un document XML.
- La méthode getElementById est basée sur l'attribut spécial xml:id (de type ID dans une DTD).

```
<voiture xml:id="voiture1">...</voiture>
<voiture xml:id="voiture2">...</voiture>
```

- La méthode `getElementById("code")` de la classe Document trouve l'élément portant l'attribut `xml:id="code"` ou null s'il n'y en a pas dans le document.
- Par exemple, on cherche la voiture2 :

```
Element voiture2 = document.getElementById("voiture2");
```

- On peut ensuite directement traiter l'élément (sauf si null).

Traitement d'un noeud

- On étudie maintenant ce qui est fait dans le coeur de la boucle des algorithmes précédents.
- D'abord faire attention, ce ne sont pas forcément que des instances d'Element, ça peut être des commentaires, des textes ou d'autres noeuds. Il faut donc faire un test sur le type de noeud :

```
Node courant = ......;
// traiter le noeud courant
switch (courant.getNodeType()) {
   case Node.ELEMENT_NODE: // c'est un élément
        break;
   case Node.TEXT_NODE: // c'est un texte
        break;
   case Node.COMMENT_NODE: // c'est un commentaire
        break;
   ...
```

Traitement d'un élément

- Dans la pratique, on se contente des tests qui nous intéressent afin d'extraire les données dont on a besoin. Par exemple :

```
Node courant = ...
// traiter le noeud courant
if (courant.getNodeType() == Node.ELEMENT_NODE &&
   courant.getNodeName().equals("voiture")) {
   // on est sur un élément <voiture>
   Element voiture = (Element) courant;
   // traiter cet élément
   ...
   }
```

- La conversion du Node en Element permet d'utiliser les getters spécifiques pour avoir ses attributs ou son contenu.

Contenu d'un noeud texte

- Soit un Element représentant la marque de la voiture, correspondant à `<marque> Renault</marque>`
- Comment faire pour récupérer le contenu texte, "Renault" de cet élément ?
- On utilise la méthode `getTextContent()` :

```
// on est sur un élément <marque>
Element marque = (Element) courant;
String texte = marque.getTextContent();
```

- **Important:** il faut savoir que `getTextContent()` concatène tous les textes contenus dans l'élément et tous ses sous-éléments.

```
<Person>
    <Firstname>Bensalem</Firstname>
    <Lastname>Mohamed</Lastname>
</Person>
```

BensalemMohamed

Création d'un document XML

```
package test1;
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Comment;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Creation_XML {

public static void main(String[] args) {
try {
        // création d'une fabrique de parseurs (Factory)
        DocumentBuilderFactory factory= DocumentBuilderFactory.newInstance();
        // création d'un parseur (Builder)
        DocumentBuilder builder= factory.newDocumentBuilder();
        // création d'un document
        Document document = builder.newDocument();
```

Création d'un document XML

```
// création de la racine du document
Element racine = document.createElement("voiture");

// Ajout d'un attribut
racine.setAttribute("Matricule", "1970-122-22");

 // ajout de la racine au document
document.appendChild(racine);

// Création d'un élément
Element marque = document.createElement("marque");

// ajout de texte à l'élément
marque.appendChild(document.createTextNode("Renault"));

// ajout de l'élément sous la racine
racine.appendChild(marque);

 // ajout d'un commentaire
Comment commentaire = document.createComment("ceci est un commentaire");
racine.appendChild(commentaire);
```

Création d'un document XML

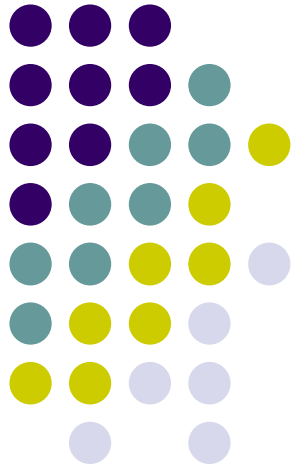
```
        // création de la fabrique
        TransformerFactory transformerFactory = TransformerFactory.newInstance();

        // récupération du transformeur
        Transformer transformer =  transformerFactory.newTransformer();

        // configuration du transformeur
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

        // écriture du document dans un fichier
        DOMSource source = new DOMSource(document);
        StreamResult sortie =new StreamResult(new File("sortie.xml"));
        transformer.transform(source, sortie);

      }
        catch (Exception e){
            e.printStackTrace();
        }

}

}
```

Interfaces de Programmation pour XML

DOM & SAX

Pr. Sidi Mohammed Benslimane
École Supérieure en Informatique
08 Mai 1945 – Sidi Bel Abbes –

s.benslimane@esi-sba.dz



Présentation de l'API SAX

- SAX signifie *Simple API for XML*, mais aurait pu être appelée *Sequential Access for XML*.
- Cette interface de programmation permet de lire et traiter un document XML sans le stocker entièrement en mémoire. C'est au contraire de DOM qui stocke la totalité du document sous forme d'un arbre de Nœuds.
- SAX est destiné à traiter des documents qui sont trop gros à stocker en mémoire ou dont on n'a pas besoin de parcourir le contenu de manière aléatoire.
- SAX ne permet qu'un seul parcours du document, dans l'ordre dans lequel il a été enregistré.

Principes de SAX

- Un document est transformé en un flux d'évènements syntaxiques (balise ouvrante, balise fermante, etc.),
- Une application SAX est un écouteur (*listener*) d'évènements, possédant certaines méthodes qui sont appelées en fonction de ce qui se trouve dans le document XML. ***C'est de la programmation événementielle***.
- C'est comme avec les interfaces Swing, vous définissez un écouteur pour les clics souris. Lorsque l'utilisateur clique, cela appelle la méthode que vous avez définie.

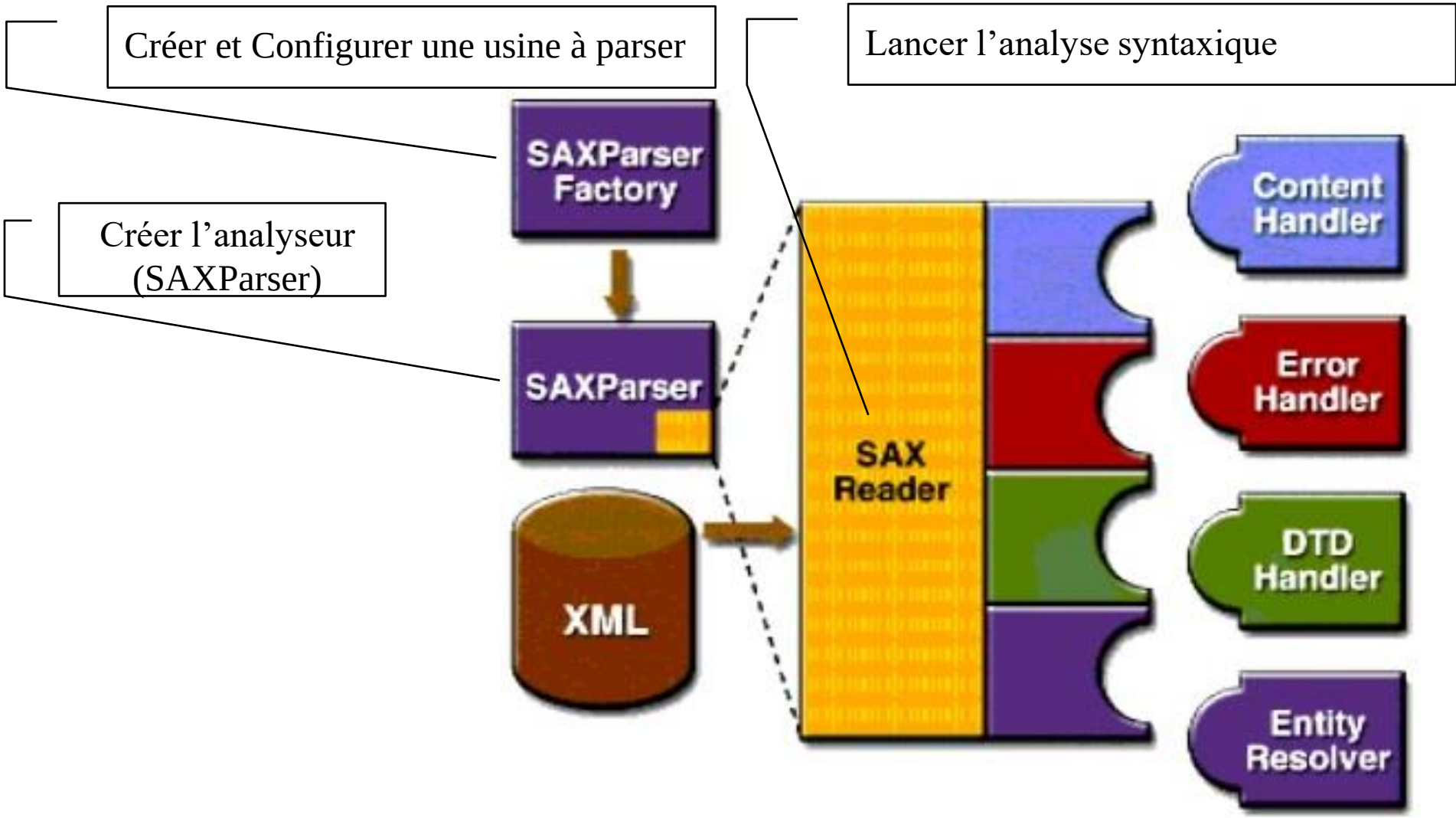
Fonctionnement de SAX

- SAX parcourt le document et le découpe en fragments : balises ouvrantes, balises fermantes, textes, etc. À chaque fragment rencontré, il appelle une méthode spécifique de l'écouteur.

```
<?xml version="1.0" encoding="UTF-8"?>
<Cours>
    <Titre>Cours SAX</Titre>
    <Auteur>
        <Nom>BENSLIMANE</Nom>
        <Prénom>Sidi Mohamed</Prénom>
    </Auteur>
    <Description>
        Ce cours aborde les <b>concepts</b> de
        base mis en &#339;uvre dans SAX.
    </Description>
</Cours>
```

- **startDocument**
- **startElement: Cours**
- **startElement: Titre**
- **characters: "Cours SAX"**
- **endElement: Titre**
- **startElement: Auteur**
- **startElement: Nom**
- **characters: "BENSLIMANE"**
- **endElement: Nom**
- **startElement: Prénom**
- **characters: "Sidi MOhamed"**
- **endElement: Prénom**
- **endElement: Auteur**
- **startElement: Description**
- **characters: "Ce cours aborde les "**
- **startElement: b**
- **characters: "concepts"**
- **endElement: b**
- **characters: " de base mis en oeuvre dans XML."**
- **endElement: Description**
- **endElement: Cours**
- **endDocument**

Les quatre interfaces de SAX



Les quatre interfaces sont implantées par **`DefaultHandler`**

Interface ContentHandler

- Content Handler est l'interface centrale 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, textes, commentaires, etc.).
- Cette interface définit ce que doit implémenter un écouteur SAX. Ce sont 11 méthodes, dont les plus importantes sont:

Interface ContentHandler

- `startDocument()` : Notification du début du document.
- `endDocument()` : Notification de la fin du document.
- `startElement( String uri, String localName, String qName, Attributes attrs)` : Notification du début d'un élément,
- `endElement( String uri, String localName, String qName)` : Notification de la fin d'un élément,
- `characters( char[] ch, int start, int length)` : Signale une zone de texte qui est à extraire du tableau ch

Interface ContentHandler

startElement(String uri, String localName, String qName, Attributes attrs)

- **String uri** : l'URI de l'espace de nom de cet élément ;
- **String localName** : le nom local de cet élément, c'est-à-dire sans son préfixe ni son espace de noms ;
- **String qName** : le nom qualifié de cet élément, avec son préfixe ;
- **Attributes attributes** : les attributs de cet élément, s'il en a.

Exemple

<galilee:marins xmlns:galilee="http://www.galilee.org/marins" lang="FR"**/>**

- **uri** : http://www.galilee.org/marins
- **localname** : **marins**
- **qName** : **galilee:marins**
- **Attributes** : lang="FR"

Interface ContentHandler

```
startElement( String uri, String localName,
String qName, Attributes attrs)
```

- Le type `Attributes` mentionné dans `startElement` représente un tableau d'attributs :
- `String getLength()` retourne le nombre d'attributs
- `String getValue(String nomqual)` retourne la valeur de l'attribut ayant ce nom qualifié
- `String getQName(int i)` retourne le nom qualifié du $i^{ième}$ attribut.

Interface ErrorHandler

- Le gestionnaire d'erreur, bien que facultatif, est très utile pour vérifier qu'un document est bien formé et valide. Cette interface propose trois méthodes:
  - Warning: ce n'est pas en soi une erreur, mais l'indication d'une incohérence, comme la présence de définitions inutilisées dans une DTD, ou par exemple un caractère inconnu.
    - `void warning(SAXParseException exception)`
  - Erreur : il s'agit d'une erreur de validation liée à un schéma ou une DTD.
    - `void error(SAXParseException exception)`
  - Erreur fatale : il s'agit d'une erreur de syntaxe qui ne permet plus au parseur de continuer son travail. par exemple le fichier n'est pas lisible.
    - `void fatalError(SAXParseException exception)`

Gestion des erreurs

- La classe **`DefaultHandler`** implémente l'interface **`ErrorHandler`** qui récupère les exceptions provoquées par les erreurs.

```
class MonHandler extends DefaultHandler {
        ...
        public void fatalError(SAXParseException e) {
            System.err.println( "Erreur fatale ligne" +e.getLineNumber()
            + " colonne " +e.getColumnNumber());
            System.err.println(e.getMessage());
         }
}
```

- Après une erreur fatale, l'analyse s'arrête définitivement.

Interface DTDHandler et EntityResolver

- L'interface DTDHandler est liée à l'analyse de la DTD par le parseur.
- L'interface EntityResolver est utilisée pour effectuer un traitement lors de la résolution d'entité.

Programmation d’un analyseur

Implémentation d'un Analyseur

- Implémentation d'un Analyseur passe par les quatre étapes suivantes:

1. Créer une usine à parser (SAXParserFactory)
2. Créer l'analyseur (SAXParser)
3. Créer un écouteur qui sera activé par l'analyseur (DefaultHandler)
4. Lancer l'analyse syntaxique (méthode parse)

Implémentation d'un Analyseur

```
void Analyser(String documentURL) throws Exception {
        // 1. créer un générateur d'analyseur
        SAXParserFactory factory = SAXParserFactory.newInstance();
        // 2. créer un analyseur
        SAXParser parser = factory.newSAXParser();
        // 3. créer un écouteur qui sera activé par l'analyseur
        MonHandler handler = new MonHandler();
        // 4. lancer l'analyse sur l'URI : fichier ou http://...
        parser.parse(documentURL, handler);
}
```

- documentURL est le nom d'un fichier ou une URL sur le réseau.

Implémentation d'un ContentHandler

- Pour traiter la plupart des documents, on peut se contenter de définir **`startElement`**, **`endElement`** et **`characters`** et dériver la classe **`DefaultHandler`** qui implémente **`ContentHandler`**

```
class MonHandler extends DefaultHandler {
        public void startElement(...) throws SAXException {
        ...
        }
        public void endElement(...) throws SAXException {
        ...
        }
        public void characters(char[] text, int debut, int lng) {
        String texte = new String(text, debut, lng);
        ...
        }
}
```

Fonctionnement d'un analyseur

- 1. Ouverture du fichier à analyser
- 2. Lecture d'une partie significative du fichier :
- 3. Si cette partie:
  a) est une balise ouvrante, balise fermante, du texte, etc., appel d'une méthode du **`ContentHandler`**
  b) est un composant de la DTD, appel d'une méthode du **`DTDHandler`**
  c) est une référence à une entité, appel d'une méthode du **`EntityResolver`**
  d) pose un problème (document mal formé ou non valide), appel d'une méthode du **`ErrorHandler`**
- 4. Si le document n'est pas terminé, retour en 2

Exemple 1 d'un analyseur

```
public class Lire_XML_SAX extends DefaultHandler {
    public void startDocument() throws SAXException {
        System.out.println("Debut du Document");
    }
    public void startElement(String uri, String localName, String qName,
            Attributes attributes) throws SAXException {
        System.out.println("Debut de L'Element : " + qName);
    }
    public void characters(char[] ch, int start, int length)
            throws SAXException {
            System.out.println(new String(ch, start, length));
                    }
    public void endElement(String uri, String localName, String qName)
            throws SAXException {
        System.out.println(" Fin de L'Element : "+ qName);
    }
    public void endDocument() throws SAXException {
        System.out.println("Fin du Document");
    }
    public static void main(String[] args) throws ParserConfigurationException,
    SAXException, IOException {
    // créer un générateur d'analyseur
    SAXParserFactory factoy = SAXParserFactory.newInstance();
    // créer un analyseur
    SAXParser parser = factoy.newSAXParser();
    // lancer l'analyse sur le ficher livre.xml
    File xmlFile = new File("livres.xml");
    parser.parse(xmlFile, new Lire_XML_SAX());
    }
}
```

Exemple 2 d'un analyseur

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document; import org.w3c.dom.Element; import org.w3c.dom.Node;

public class Lecture_XML {

          public static void main(String[] args) {

          try {
          DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
          DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
          Document doc = dBuilder.parse("eleve.xml");
          Element rootElement = doc.getDocumentElement();
          Node courant = rootElement.getFirstChild();
          while (courant != null) {
                                        // traiter le noeud courant
                                        String nom = courant.getNodeName();
                                        String contenu = courant.getTextContent();
                                        System.out.println(nom + ": " + contenu + "  ");
                                        // passer au suivant
                                        courant = courant.getNextSibling();
                              }
                    } catch (Exception e) {
                              e.printStackTrace();
                    }

          }

}
```

Traitement d’un document XML

Aucune visibilité globale

- La conséquence du fonctionnement événementiel de SAX, c'est qu'on n'a aucune vision globale du document.

```
<voiture id="871">
    <prix monnaie="euros">13312</prix>
</voiture>
```

- Voici les événements déclenchés en séquence mais indépendamment les uns des autres :
  - `startElement("","voiture","voiture",[id="871"])`
  - `startElement("","prix","prix",[monnaie="euros"])`
  - `characters("13312")`
  - `endElement("","prix","prix")`
  - `endElement("","voiture","voiture")`
- Quand on est dans la méthode characters, on ne dispose plus des attributs de la balise ouvrante prix.
- Comment faire pour convertir le prix en Dinar et l'associer à la voiture ?

Mémoriser les informations au passage

- Le principe est de mémoriser certaines informations pendant le parcours des données :

- Il faut mémoriser les informations dont on a besoin. Par exemple, quand on rencontre l'élément <prix>, il faut mémoriser la valeur de l'attribut monnaie, afin de pouvoir faire la conversion au moment où on rencontrera le texte de la valeur.

- Il faut aussi gérer un *état* indiquant où on se trouve dans l'arbre XML sous-jacent, par exemple, pour savoir quand on est dans le texte de l'élément <prix> parce que tous les textes sont gérés par la même méthode characters.

Traitements des transitions

- Par exemple, voici une partie du traitement de l'événement **`startElement`** :

```
private double m_TauxChange;
public void startElement(String uri, String name, String qName,
                                Attributes attrs) {
        // calculer le taux de change
        if (qName.equals("prix")) {
            String monnaie = attrs.getValue("monnaie");
            switch (monnaie) {
                case "euros" :m_TauxChange = 120.5 ; break;
                case "dolars":m_TauxChange = 101.5 ; break;
                case "DZ"    :m_TauxChange = 1.0 ; break;
               ...
                                }
                                    }
                                            }
```

Traitements des transitions (suite)

- Pour le traitement des événements character, on se contente de mémoriser le texte dans une variable globale. Le traitement de ce texte sera fait dans l'événement endElement (c'est un choix !, on pourrait faire autrement).

```
private String m_Texte;
public void characters(char[] text, int debut, int lng)
{
    m_Texte = new String(text, debut, lng);
}
```

Traitements des transitions (fin)

- Pour finir, voici le traitement de l'événement endElement. C'est lui qui affiche le prix en DZ:

```
public void endElement(String uri, String name, String qName) {
        if (qName.equals("prix")) {
                double prix = Float.valueOf(m_Texte) * m_TauxChange;
                System.out.println("prix = "+prix+" DZ");
                }
        }
```

DOM Vs SAX

DOM	SAX
Un processus qui utilise le DOM ne peut traiter l'arbre qu'après la lecture entière du document	Un analyseur SAX délivre les données à un processus au fur et à mesure de la lecture du document
Accès aléatoire	Accès séquentiel
Utilise beaucoup de mémoire	Utilise peu de mémoire
Construction de l'arbre du document	Événementiel
Richesse des fonctionnalités	Fonctionnalités rudimentaires
Programmation aisée	Beaucoup de code à produire
Permet la lecture et l'écriture	Permet seulement la lecture
Utile pour les petits et moyens documents XML	Utile pour les grands documents XML