

XML et Programmation Internet

Cours 6

kn@lri.fr

Plan

1 Introduction, UTF-8 et XML ✓

2 XPath ✓

3 XPath (suite) ✓

4 XSLT ✓

5 XSLT (suite) ✓

6 DOM

6.1 Le modèle DOM

6.2 Java API for XML Processing

Programmer avec XML

La représentation textuelles de documents XML n'est pas adaptée à la manipulation des données par un programme :

- On ne veut pas lire le fichier « caractère par caractère »
- On veut s'assurer que le fichier est bien formé et valide
- On veut pouvoir manipuler la structure d'arbre que représente le fichier

Document Object Model

DOM est une **spécification** du W3C qui explique **comment** représenter un document dans un langage **orienté objet**.

Avantages :

- N'est pas limité à un seul langage
- Permet de spécifier une API unique : programmer en XML en Java ou Python ne sera pas différent

Inconvénients :

- En pratique, orienté Java
- Se focalise sur les lanages objets de manière arbitraire

Que définit le DOM ?

Le DOM définit des **interfaces** (c'est à dire, **des noms de classes** auxquels sont associés des **propriétés**). Il définit aussi des types de bases (chaînes de caractères, entiers, etc.) et des types auxiliaires qui sont implantés par les types de bases du langage.

L'interface Node (1/4, constantes)

```
//attention ce n'est pas du Java
interface Node {

//constantes entières définissant les types de nœuds
const unsigned short    ELEMENT_NODE           = 1;
const unsigned short    ATTRIBUTE_NODE        = 2;
const unsigned short    TEXT_NODE             = 3;
const unsigned short    CDATA_SECTION_NODE    = 4;
const unsigned short    ENTITY_REFERENCE_NODE = 5;
const unsigned short    ENTITY_NODE          = 6;
const unsigned short    PROCESSING_INSTRUCTION_NODE = 7;
const unsigned short    COMMENT_NODE         = 8;
const unsigned short    DOCUMENT_NODE        = 9;
const unsigned short    DOCUMENT_TYPE_NODE   = 10;
const unsigned short    DOCUMENT_FRAGMENT_NODE = 11;
const unsigned short    NOTATION_NODE       = 12;
```

L'interface Node (2/4, valeur, nom et type)

```
//nom et valeur du nœud
```

```
readonly attribute DOMString      nodeName;  
attribute DOMString               nodeValue;
```

- **Pour les éléments** nodeValue **vaut** null **et** nodeName **est le nom de la balise**
- **Pour les nœuds texte** nodeValue **est le texte et** nodeName **est la chaîne fixe** #text
- **Pour les attributs** nodeValue **vaut la valeur de l'attribut et** nodeName **est son nom**

```
//L'une des 12 constantes du slide précédent  
readonly attribute unsigned short  nodeType;
```

L'interface Node (3/4, navigation)

```
readonly attribute Node           parentNode;
readonly attribute NodeList       childNodes;
readonly attribute Node          firstChild;
readonly attribute Node          lastChild;
readonly attribute Node          previousSibling;
readonly attribute Node          nextSibling;
readonly attribute NamedNodeMap  attributes;
```

Utilise deux interfaces auxiliaires:

```
interface NodeList {
Node          item(in unsigned long index);
readonly attribute unsigned long  length;
};
interface NamedNodeMap {
Node          getNamedItem(in DOMString name);
...
}
```


L'interface Node (4/4, mise à jour)

```
//Renvoie le document auquel appartient le nœud
readonly attribute Document      ownerDocument;

Node      insertBefore(in Node newChild,
                      in Node refChild)
                      raises(DOMException);

Node      replaceChild(in Node newChild,
                      in Node oldChild)
                      raises(DOMException);

Node      removeChild(in Node oldChild)
                      raises(DOMException);

Node      appendChild(in Node newChild)
                      raises(DOMException);

boolean   hasChildNodes();

//Nécessaire pour copier un nœud d'un document dans un autre
Node      cloneNode(in boolean deep);
```

Sous-interfaces de Node

L'interface `Node` est spécialisée en 12 sous-interfaces différents (les 12 types de nœuds possibles). Les principales sont:

- `Document` : l'interface du nœud « racine » du document
- `Element` : l'interface des nœuds correspondant à des balises
- `Attr` : l'interface des nœuds correspondant à des attributs
- `Text` : l'interface des nœuds correspondants à des textes

L'interface Text

```
interface Text : Node {  
  //renvoie vrai si le nœud ne contient que des espaces  
  readonly attribute boolean      isElementContentWhitespace;  
  ...  
}
```

(La spécification de DOM mentionne d'autres propriétés)

L'interface Attr

```
interface Attr : Node {  
  readonly attribute DOMString      name;  
  readonly attribute DOMString      value;  
  
  readonly attribute Element         ownerElement;  
  ...  
};
```

L'interface Element

```
interface Element : Node {
readonly attribute DOMString      tagName;
//manipulation par chaine :
DOMString      getAttribute(in DOMString name);
void           setAttribute(in DOMString name,
                           in DOMString value)
                           raises(DOMException);

void           removeAttribute(in DOMString name)
                           raises(DOMException);

//manipulation par nœud :
Attr           getAttributeNode(in DOMString name);
Attr           setAttributeNode(in Attr newAttr)
                           raises(DOMException);
Attr           removeAttributeNode(in Attr oldAttr)
                           raises(DOMException);

//renvoie tous les descendants avec un certain tag
NodeList      getElementsByTagName(in DOMString name);
}
}
```

L'interface Document

```
interface Document : Node {
//L'élément racine
readonly attribute Element          documentElement;

//Création de nœuds pour ce document :

Element          createElement(in DOMString tagName)
                    raises(DOMException);

Text             createTextNode(in DOMString data);
Attr             createAttribute(in DOMString name)
                    raises(DOMException);

//Les descendants avec un tag particulier
NodeList         getElementsByTagName(in DOMString tagname);

//Copie un nœud, éventuellement avec ses descendants et
//en fait un nœud ajoutable au document :
Node             importNode(in Node importedNode,
                            in boolean deep);
```

Modèle mémoire

Un **nœud** (objet implémentant l'interface Node) ne peut pas appartenir à deux documents. Exemple :

```
Node noeud_a = document1.getElementByTagName("a").item(0);  
  
document2.appendChild(noeud_a); //Exception si document2 n'est  
                                //pas le même objet que  
                                document1  
  
//par contre ceci est ok:  
document2.appendChild(document2.importNode(noeud_a, true));
```

Plan

1 Introduction, UTF-8 et XML ✓

2 XPath ✓

3 XPath (suite) ✓

4 XSLT ✓

5 XSLT (suite) ✓

6 DOM

6.1 Le modèle DOM ✓

6.2 Java API for XML Processing

Introduction à JAXP

API de la bibliothèque standard Java qui permet de manipuler du XML. Elle comprend (entre autres) :

- Lecture et écriture de documents **en streaming** (cours 7)
- Implémentation complète de la spécification **DOM**
- Moteur XSLT (et donc XPath)

Inconvénients : la bibliothèque essaye d'être très générique, afin que n'importe qui puisse fournir son implémentation de DOM en utilisant les interfaces fournies. Il faut donc parfois passer par des grandes séquences d'incantations magiques pour créer un objet

Structure de l'API

- Les types et interfaces spécifiés par le w3C se trouvent dans le packages `org.w3c.*`
- Les types « usuels » pour XML sont dans `org.xml.*`
- Les **classes** concrètes java implémentant les interfaces sont dans `javax.xml.*`

Factory *design pattern*

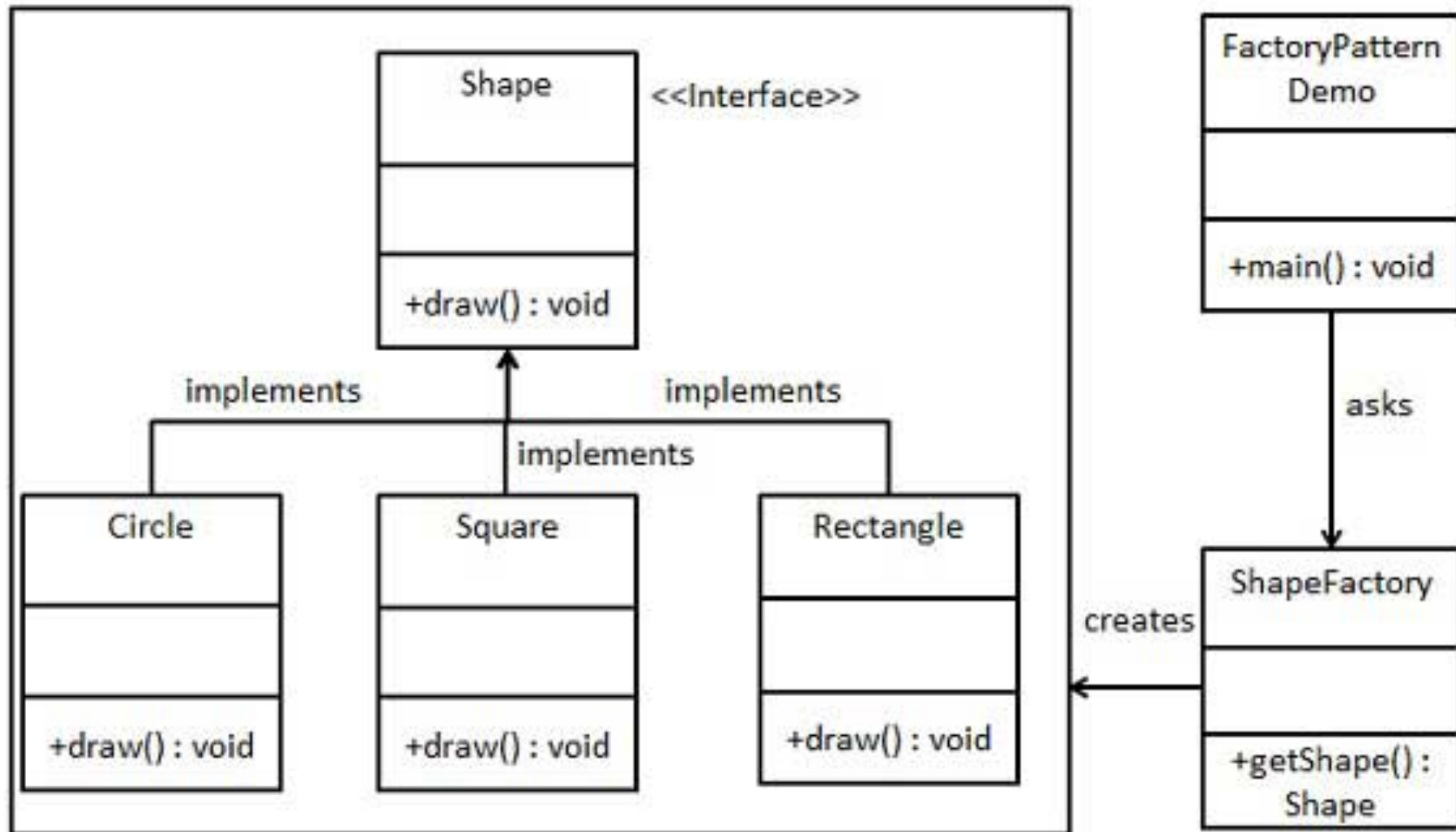
Comme toutes les API complexes en Java (et dans les langages objets en général), Jaxp utilise le *design pattern* de **Factory**.

Pour créer un objet de type `Foo` on ne fait pas simplement `new Foo(...)`; mais on utilise une classe `FooFactory` qui possède une méthode `.createFoo(...)`

Dans quel cas est-ce intéressant ?

Quand `Foo` est une interface. On ne peut pas faire `new` sur une interface. Il faut donc une méthode pour appeler le constructeur de la classe implémentant `Foo` puis qui le caste en `Foo` ...

Factory design pattern (exemple)



Création d'un document : DocumentBuilder

La classe `DocumentBuilder` permet de créer un document XML :

- Soit en le lisant depuis un fichier (avec la méthode `parse()`)
- Soit vide, avec la méthode `newDocument()`

Pour obtenir un `DocumentBuilder`, il faut passer par un `DocumentBuilderFactory` :

```
//création de la Factory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

//On définit quelques options
dbf.setIgnoringElementContentWhitespace(true); // option
dbf.setValidating(false); // option

//On crée le documentBuilder
DocumentBuilder db = dbf.newDocumentBuilder();

//On charge le document
Document doc = db.parse("fichier.xml");
```

DOM dans JAXP

Le DocumentBuilder permet d'obtenir un Document (interface Java qui implémente l'interface DOM du même nom)

Conventions de nommage : les *propriétés* des interfaces DOM sont préfixées par **get** ou **set** en Java. Les méthodes ont le même nom. Exemple :

```
Node n = ...;  
n.getNodeType(); //DOM défini nodeType;  
n.getFirstChild(); //DOM défini firstChild;  
n.appendChild(m); //C'est une méthode en DOM donc même nom
```

Conversions

On travaille la plupart du temps avec des objets ayant le type `Node`. La manière correcte de les convertir est la suivante :

```
switch (n.getNodeType()) {  
  
case Node.DOCUMENT_NODE:  
    Document d = (Document) n;  
    ...  
    break;  
case Node.ELEMENT_NODE:  
    Element e = (Element) n;  
    ...  
    break;  
case Node.TEXT_NODE:  
    Text t = (Text) n;  
    ...  
    break;  
}
```

Rappels : classes et interfaces utiles en Java

Interface `Map<K, V>` permet d'associer des clés de types `K` à des valeurs de type `V` (`K` et `V` doivent être des `Objects` donc pas `int`, `bool`, ...)

Implémentations possibles : `TreeMap`, `HashMap`