XML - Semaine 6

Pierre Nerzic

février-mars 2022

Le cours de cette semaine présente l'API XML DOM permettant de produire et traiter un document XML par programme. Cette API sera étudiée dans son implémentation Java, mais elle est disponible dans quasiment tous les langages.

Plan du cours :

- Principes,
- Création et modification d'un XML,
- Lecture et traitement d'un XML,
- Sérialisation d'objets Java en XML,
- XML DOM dans d'autres langages.

Principes

Pierre Nerzic

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.

L'API DOM est définie par le W3C, c'est à dire le *World Wide Web Consortium* qui normalise tout ce qui concerne le Web, dont XML.

Le sigle DOM signifie *Document Object Model*. Cette API manipule une représentation d'un document complet. La totalité du document est chargée en mémoire pendant le traitement.

C'est cette API qui est utilisée en JavaScript dans un document HTML : `document.getElementsByTagName("div")`, etc.

Pierre Nerzic

Présentation, suite

Il existe une autre API appelée SAX (*Simple API for XML*) qui permet de lire un document XML de manière séquentielle sans rien stocker en mémoire. Voir le prochain cours.

Pierre Nerzic

Principe généraux de l'API DOM

L'API W3C DOM est utilisable avec de nombreux langages : Java, JavaScript, PHP, Python, C++... Quand on crée ou qu'on ouvre un document XML, un objet `Document` est créé pour représenter le document tout entier. Les méthodes de cette instance permettent de créer ou parcourir les éléments, attributs et textes du document.

- En mode création :
  1. créer une instance de `Document` (vide)
  2. ajouter des instances d'`Element` au document,
     a. ajouter des sous-éléments, attributs, textes, CDATA...
  3. écrire le document dans un fichier ou sur le réseau.
- En mode lecture d'un fichier :
  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.

Bibliothèques

Pour travailler avec l'API, il faut importer un petit nombre de librairies, toujours les mêmes :

```
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;
```


Pierre Nerzic

Document DOM en mode création

Pierre Nerzic

Création d'un objet `Document`

En Java, il faut trois instructions :

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();
```

1. Création d'une `factory` : c'est un singleton qui permet de créer des objets d'un certain type, ici des `DocumentBuilder`.
2. Création d'un `builder` : encore un singleton mais spécialisé dans la création de documents XML.
3. Création d'un `document` : c'est lui qui représente le document XML qu'on veut manipuler.

Pierre Nerzic

Compléments

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 = DBF....newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.newDocument();
    // remplir le document avec des éléments, etc.
    ...
  } catch (Exception e) {...}
}
```

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;

// écrivain
TransformerFactory transformerFactory =
    TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();

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

Ajout d'éléments au document

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);`

```
import org.w3c.dom.Element;

// 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);
```

Insertion avant un autre élément

La méthode `parent.appendChild(element)` ajoute l'élément à la fin de la liste du parent.

On peut insérer un élément avant un autre avec : 

```
Element prix = document.createElement("prix");
racine.insertBefore(prix, marque);
```

Cela va insérer `<prix>` avant `<marque>` dans la racine.

NB: le nœud parent d'un élément s'obtient par : 

```
Element parent = (Element) element.getParentNode();
```


Pierre Nerzic

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"));
```

Mais on a aucune variable pour représenter les éléments rajoutés, on ne peut 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 encore plus simplement :

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

Notez que les noms et valeurs sont des chaînes. Si vous avez des nombres à affecter, il faudra les convertir en textes avec `String.valueOf(`*nombre*`)`.

Espaces de nommage

Lorsqu'un élément doit avoir un *namespace* identifié par un *URI* et un préfixe, il faut créer l'élément avec la méthode `createElementNS(`*URI*`,`*nom qualifié*`)` :

Rappel du cours 1 : le nom qualifié est composé d'un préfixe et d'un nom local séparés par : 

```
final static String URI = "urn:iutlan:test";
final static String PREFIXE = "iutlan:";
Element element =
    document.createElementNS(URI, PREFIXE+"element");
```

De même avec les attributs : 

```
element.setAttributeNS(URI, PREFIXE+"attribut", "valeur");
```


Ajout de textes et de CDATA

Nous arrivons au contenu d'un élément. Il est très simple de rajouter du texte dans un élément. Il n'est pas forcément nécessaire d'associer une variable sauf si le texte doit être modifié ultérieurement Il y a deux possibilités :

```
element.appendChild(document.createTextNode("texte"));
```

Et en beaucoup plus simple :

```
element.setTextContent("texte");
```

On peut aussi rajouter des sections CDATA par :

```
element.appendChild(document.createCDATASection("data"));
```

NB: les sections CDATA sont des nœuds frères des textes et non pas des nœuds enfants (ils sont au même niveau).

Ajout de commentaires et autres

C'est aussi simple que de rajouter du texte : 

```
element.appendChild(document.createComment("commentaire"));
```

Il y a également une méthode pour créer une instruction de traitement, c'est à dire un nœud `<?action parametres?>` : 

```
racine.appendChild(
    document.createProcessingInstruction(
        "xml-stylesheet", "href='style.xsl' type='text/xsl'"));
```

Classe Node

Les classes `Element`, `TextNode`, `Comment`... sont toutes des sous-classes de `Node`[1]. Un `Node` représente l'un des nœuds de l'arbre XML sous-jacent (voir cours 1).

Dans le modèle W3C, un `Node` possède un *type*. C'est un petit entier `short` retourné par la méthode `getNodeType()`. Des constantes permettent de nommer ces types :

`Node.ELEMENT_NODE` pour les `Node` de type `Element`
`Node.TEXT_NODE` pour les `Node` de type `Text`
`Node.DOCUMENT_NODE` pour les `Node` de type `Document`
`Node.ATTRIBUTE_NODE` pour les `Node` de type `Attr`
`Node.COMMENT_NODE` pour les `Node` de type `Comment`

[1] En réalité, en Java, ce sont des interfaces et non pas des classes.

Document DOM en mode lecture

Pierre Nerzic

Traitement du document

On se place maintenant du côté lecture et analyse d'un document XML existant. En général, on a besoin de :

- chercher un ou plusieurs nœuds spécifiques,
- itérer sur tous les nœuds enfants d'un nœud,
- vérifier le nom d'un nœud,
- extraire les valeurs d'attributs ou le contenu texte d'un nœud.

C'est souvent un ensemble de tout cela.

Ouverture d'un fichier

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

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true); // s'il y a des namespaces
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("document.xml"));
```

Notez les deux changements :

- (ligne 2, optionnelle) Il va y avoir des *namespaces*,
- (ligne 4) On remplit le document avec ce qui se trouve dans le fichier XML.

En fait, on peut aussi compléter ou modifier le document existant puis l'enregistrer comme dans la partie précédente.

Prologue du document

Des méthodes de `Document` permettent d'obtenir les informations du prologue :

- `String` *document*`.getXmlVersion()` retourne la version, c'est `"1.0"` en général.
- `String` *document*`.getXmlEncoding()` retourne l'encodage, par exemple `"UTF-8"`.

NB: il n'est pas du tout nécessaire de récupérer ces informations pour traiter le document.

Il faut également noter que les *setters* existent pour configurer un document en création/modification. Par exemple *document*`.setXmlVersion("1.1");`

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.

C'est une instance de la classe `Element`. Puis pour avoir le nom de la racine, on emploie l'un de ses *getters* :

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

Dans un programme, on se contente en général de vérifier que la racine porte le bon nom.

Espaces de nommages

Le nom d'un élément s'obtient par `getNodeName()` ou `getTagName()` qui est équivalente.

Lorsqu'il y a un *namespace*, le nom de l'élément s'appelle un *nom qualifié*, c'est ce qui est retourné par les deux méthodes précédentes, et il est composé d'un *préfixe* séparé du *nom local* par un « : »

On peut obtenir :

- le préfixe : `String` *element*`.getPrefix()`
- le nom local : `String` *element*`.getLocalName()`
- l'URI du préfixe : `String` *element*`.getNamespaceURI()`

NB: toutes ces méthodes renvoient `null` si on a oublié de mettre `factory.setNamespaceAware(true);` avant de charger le fichier.

Attributs d'un `Element`

Les méthodes suivantes permettent d'obtenir les attributs d'un élément :

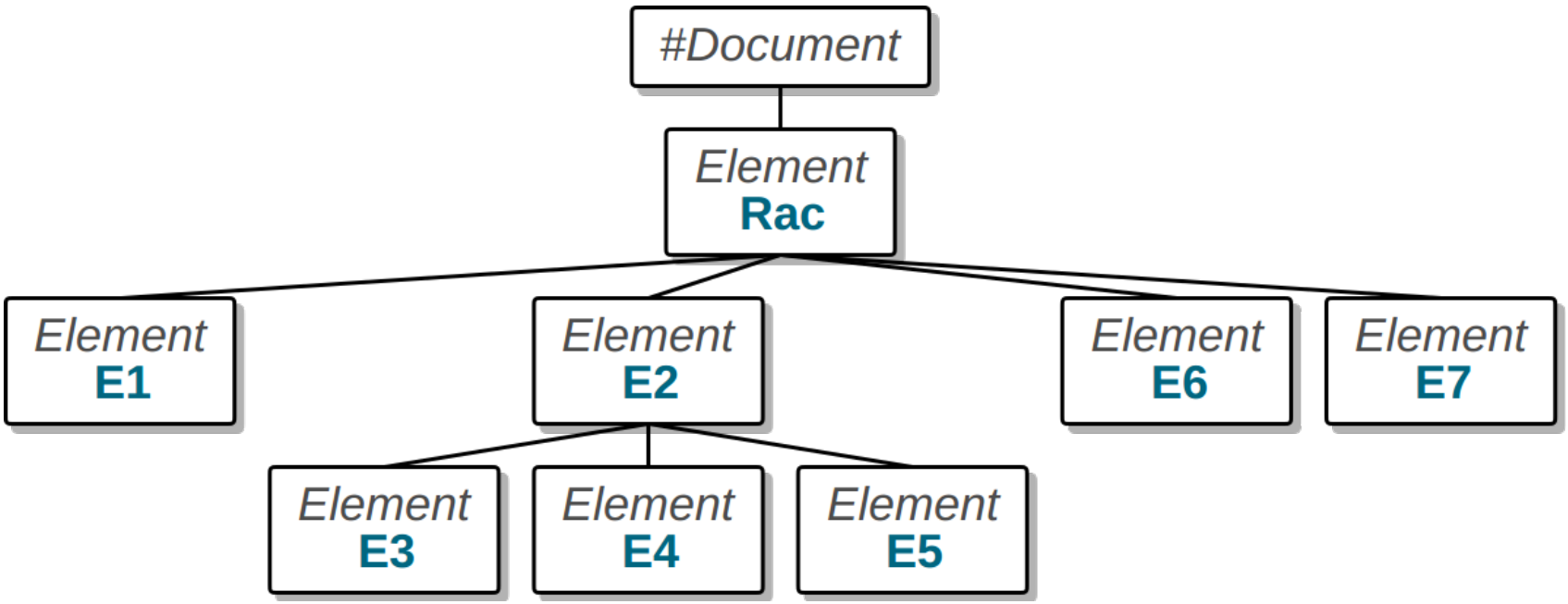
- `String` *element*`.getAttribute(`*nomattr*`)` retourne l'attribut ou la chaîne vide s'il n'y a pas cet attribut. C'est pour distinguer la présence d'un attribut qui serait vide de son absence qu'il faut tester auparavant avec la méthode suivante,
- `boolean` *element*`.hasAttribute(`*nomattr*`)` renvoie `true` si l'élément possède cet attribut

Il y a des méthodes pour tenir compte des *namespaces* des attributs. Il faut leur fournir l'URI qui définit le *namespace* :

- `String` *element*`.getAttributeNS(`*URI*`,`*nomlocal*`)`
- `boolean` *element*`.hasAttributeNS(`*URI*`,`*nomlocal*`)`

Nœuds enfants d'un élément

Une instance de la classe `Element` telle que la racine du document peut avoir un `Node` parent, des enfants, ainsi que des frères. Voici un schéma pour le transparent suivant :



Voisinage d'un nœud

Quand on considère le nœud `E2` :

- Le nœud parent de `E2` est `Rac`
  - On l'obtient par `E2.getParentNode()`
- Le précédent nœud frère de `E2` est `E1`
  - On l'obtient par `E2.getPreviousSibling()`
- Le nœud frère suivant est `E6`
  - On l'obtient par `E2.getNextSibling()`
- Le premier nœud enfant de `E2` est `E3`
  - On l'obtient par `E2.getFirstChild()`
- Le dernier nœud enfant de `E2` est `E5`
  - On l'obtient par `E2.getLastChild()`

Toutes ces méthodes retournent `null` si aucun `Node` ne correspond.

Parcours des nœuds 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 nœuds 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, un `Node`, par son indice. Voici l'algorithme : 

```
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
    ...
}
```

Le mot clé Java `final` signifie que la variable ne changera plus après son affectation. Ça accélère un peu les boucles.

Parcours des nœuds 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
    ...
}
```

Il y a une variante avec *namespace* : `getElementsByTagNameNS`

Parcours des nœuds enfants (méthode 4)

Il existe enfin une 4e manière pour trouver directement les éléments qu'on souhaite dans un document XML. Elle 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 nœud

On étudie maintenant ce qui est fait dans le cœur de la boucle des algorithmes précédents (méthodes 1 à 3).

D'abord faire attention, ce ne sont pas forcément que des instances d'`Element`, ça peut être des commentaires, des textes ou d'autres nœuds. Il faut donc faire un test sur le type de nœud :

```
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> : convertir le type
    Element voiture = (Element) courant;
    // traiter cet élément
    ...
}
```

La conversion (*type cast*) du `Node` en `Element` permet d'utiliser les *getters* spécifiques pour avoir ses attributs ou son contenu.

Traitement d'éléments

Quand différents éléments peuvent être mélangés et arriver dans n'importe quel ordre, il faut faire une sorte d'aiguillage. Java (v≥7) permet de faire une conditionnelle multiple basée sur des chaînes, ici sur le nom de l'élément courant :

```
if (courant.getNodeType() == Node.ELEMENT_NODE) {
    Element element = (Element) courant;
    switch (element.getNodeName()) {
        case "voiture":
            ...
            break;
        case "prix":
            ...
            break;
    }
}
```

Contenu d'un nœud 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, y compris les sections CDATA et les entités replacées par leurs valeurs. Il faut s'assurer que l'élément est une feuille dans l'arbre.

Sérialisation d'objets Java

Pierre Nerzic

Principes

La sérialisation d'un objet consiste à écrire ses variables membres dans un format pouvant être transmis ou stocké, et relu pour recréer les objets à l'identique. Nous allons voir comment générer un document XML à partir d'un objet Java et inversement, instancier un objet Java par un document XML.

En anglais, la sérialisation se dit *marshalling*, voir wikipedia. Mais en Java, la *sérialisation* signifie plutôt l'extraction des données binaires présentes dans la mémoire pour représenter un objet.

Inversement, la dé-sérialisation (*unmarshalling*) consiste à instancier un objet à partir de ce qu'on lit dans le document : on reconstruit un objet à partir d'un document XML.

Pour cela, on utilise l'API DOM . Les variables membres sont écrites ou relues dans les sous-éléments ou attributs.

Exemple

Voici une classe :

```
class Voiture
{
    private int id;
    private String modele;
    private int kilometrage;
}
```

Une sérialisation XML d'une instance pourrait être :

```
<voiture id="1">
    <modele>Bugatti Chiron</modele>
    <kilometrage>12499</kilometrage>
</voiture>
```

Récursivité de la sérialisation

Lorsqu'une variable membre est un objet, lui-aussi est sérialisé :

```
class Voiture {
    private int id;
    private String modele;
    private Achat achat;
}
class Achat {
    private Date date;
    private float prix;
}
```

```
<voiture id="1">
    <modele>Bugatti Chiron</modele>
    <achat date="2022-02-07">
        <prix>3987321.98</prix>
    </achat>
</voiture>
```

Sérialisation manuelle

Principe : une méthode ressemblant à `toString` :

```
public void toXML(Document document, Node parent)
{
    // sérialisation de this dans le document
    // à l'intérieur de l'élément parent
}
```

Elle reçoit deux paramètres :

- un document XML représentant le fichier XML en construction,
- un Node parent afin d'accrocher ce qu'on va produire.

Comme `toString`, elle va écrire chaque variable membre, mais sous forme d'éléments et d'attributs dans le document.

Sérialisation manuelle, suite

- On crée d'abord l'élément racine du même nom que la classe :

```
Element elClasse = document.createElement("classe");
parent.appendChild(elClasse);
```

- On ajoute les attributs convertis en chaînes :

```
elClasse.setAttribute("id", Integer.toString(id));
```

- On ajoute les sous-éléments :

```
Element elMembre = document.createElement("membre");
elMembre.appendChild(document.createTextNode(membre));
elClasse.appendChild(elMembre);
```

- Si une variable membre est un objet, alors on fait un appel récursif à `toXML(document, elClasse)`

Exemple de sérialisation

Voici l'exemple avec la classe Voiture précédente :

```
class Voiture {
  public void toXML(Document document, Node parent) {
     // élément <voiture>
     Element elVoiture = document.createElement("voiture");
     parent.appendChild(elVoiture);
     // membre id sous la forme d'un attribut
     elVoiture.setAttribute("id", Integer.toString(id));
     // membre modele sous la forme d'un élément
     Element elModele = document.createElement("modele");
     elModele.appendChild(document.createTextNode(modele));
     elVoiture.appendChild(elModele);
     // membre achat par un appel récursif
     achat.toXML(document, elVoiture);
  }
```

Cas des collections

Lorsqu'une variable membre est une collection, il faut ajouter une boucle pour écrire ses éléments. En général, on place les éléments de la collection dans un élément XML les regroupant. Exemple :

```
class Client {
    private int id;
    private List<Voiture> voitures;
}
```

```
<client id="1">
    <voitures>
        <voiture id="1">...</voiture>
        <voiture id="3">...</voiture>
        <voiture id="7">...</voiture>
    <voitures>
</client>
```

Cas des collections, suite

Mais quand il n'y a aucun risque de confusion, on peut directement placer les items dans l'élément. Exemple :

```
class Client {
    private int id;
    private List<Voiture> voitures;
}
```

```
<client id="1">
    <voiture id="1">...</voiture>
    <voiture id="3">...</voiture>
    <voiture id="7">...</voiture>
</client>
```

Principes de désérialisation manuelle

La dé-sérialisation consiste à relire un élément XML pour en faire une instance de classe. C'est donc une méthode de type **Fabrique** (patron de conception *factory*) qui effectue cette opération.

```
public static Voiture fromXML(Node node) throws Exception
{
    // test préalable sur node : doit être un élément <voiture>

    // lire les éléments et attributs du node, voir pages svtes
    int id = ...
    String modele = ...
    Achat achat = ...

    // créer une instance si tout est ok
    return new Voiture(id, modele, achat);
}
```

Principes, suite

Le principe est de n'appeler le constructeur que si toutes les valeurs ont été correctement relues. Si une valeur n'est pas correcte, on émet une exception. On stocke les valeurs correctes dans des variables locales en attendant l'appel au constructeur.

- Au préalable, il faut vérifier la validité du node fourni :

```
// vérifier que node est un élément <voiture>
if (node.getNodeType() != Node.ELEMENT_NODE)
    throw new Exception("node n'est pas un élément");
if (! node.getNodeName().equals("voiture"))
    throw new Exception("node n'est pas un élément voiture");
```

- Puis, il est pratique de convertir le `Node` en `Element` :

```
Element elVoiture = (Element) node;
```

Principes, suite

- Les variables membres qui sont des attributs peuvent être lues facilement :

```
String marque = elVoiture.getAttribute("marque");
```

- Attention à convertir les chaînes en nombres :

```
int id = Integer.parseInt(elVoiture.getAttribute("id"));
float prix = Float.parseFloat(elVoiture.getAttribute("prix"));
```

NB: on peut aussi faire ainsi :

```
int id = Integer.valueOf(elVoiture.getAttribute("id"));
float prix = Float.valueOf(elVoiture.getAttribute("prix"));
```

mais ces méthodes `valueOf` retournent respectivement des `Integer` et `Float`, et non pas des `int` et `float`.

Principes, suite

- Les membres qui sont des sous-éléments sont plus difficiles à capturer. Il faut parcourir les Node enfants, voir page 29 :

```
Node enfant = elVoiture.getFirstChild();
while (enfant != null) {
    if (enfant.getNodeType() == Node.ELEMENT_NODE) {
        switch (enfant.getNodeName()) {
            case "modele":
                modele = enfant.getTextContent();
                break;
            case "achat":
                achat = Achat.fromXML(enfant);
                break;
        }
    }
    enfant = enfant.getNextSibling();
}
```


Pierre Nerzic

APIs de sérialisation

Présentation

Afin de ne pas réinventer la roue, la sérialisation se fait de préférence avec une API (jars et outils) existante. Il y a deux catégories :

1. Les API basées sur des annotations simples :

   Le principe est d'ajouter des annotations dans les classes à sérialiser. On désigne ce qui deviendra un élément et ce qui deviendra un attribut. Dans certains cas, l'API est capable de trouver elle-même les choses à sérialiser.

   - JAXB était l'outil standard pour sérialiser des objets Java en XML, mais semble abandonné depuis Java 9.
   - XStream, est le plus simple et direct, aucune modification requise,
   - Simple à peine plus compliqué.

Présentation, suite

2. Les API basées sur un document de liaison XML-Java :

   Le principe est de créer un document ressemblant à un schéma de validation XSD et indiquant comment chaque membre de classe est sérialisé en XML. C'est beaucoup plus polyvalent que les solutions précédentes, mais la construction du document de liaison peut être difficile.

   - JiBX serait intéressant mais il n'a pas encore été mis à jour pour Java 9+, il ne se compile pas.
   - XMLBeans semble être le meilleur (maintenu, open source, performant), mais il est trop complexe pour le peu de temps disponible pour ce cours.

Sérialisation avec l'API XStream

Pierre Nerzic

Présentation

Cette API est ultra-simple, vous n'avez rien à modifier dans vos classes, sauf vous désirez un résultat spécifique. Les variables membres sont automatiquement enregistrées ou relues dans un document XML. Il suffit seulement de programmer la lecture ou l'écriture du document XML.

Sérialisation avec XStream

Soit une classe Java quelconque, par exemple `Voiture`. Sa méthode `toXML` s'écrit ainsi avec XStream :

```
public void toXML(String filename) throws Exception
{
    // voir plus loin
    XStream xstream = createXStream();

    // sérialisation
    File output = new File(filename);
    FileWriter writer = new FileWriter(output);
    xstream.toXML(this, writer);
}
```

Pour la lancer :

```
    maVoiture.toXML("voiture.xml");
```

Désérialisation avec XStream

Pour relire le fichier XML et instancier une `Voiture` :

```
public static Voiture fromXML(String filename) throws Exception
{
    // voir plus loin
    XStream xstream = createXStream();

    // désérialisation
    File input = new File(filename);
    FileReader reader = new FileReader(input);
    return (Voiture) xstream.fromXML(reader);
}
```

Pour la lancer :

```
    Voiture maVoiture = Voiture.fromXML("voiture.xml");
```

Objet `XStream`

L'instance `XStream xstream` est au cœur du processus. Il faut la configurer. C'était très simple en Java 8, mais Java 9+ pose des restrictions très fortes sur les techniques d'introspection (ou réflexivité), c'est à dire le fait pour un programme Java d'explorer son propre code exécutable. C'est l'introspection qui permet à XStream de trouver les noms et types des variables membres à sérialiser. Java 9 bloque cette inspection pour des raisons de sécurité. Seules des classes déclarées explicitement comme « inspectables » peuvent être explorées.

On arrive donc à une initialisation assez complexe de cet objet `XStream` en Java 9+. Le fichier UtilXML.java contient le code complet.

Objet XStream, suite

Voici un résumé :

```
public static XStream createXStream() {
    // instance de XStream
    XStream xstream = new XStream(new StaxDriver(){OUT}) {CFG};
    xstream.autodetectAnnotations(true);

    // permissions, ici toutes les classes sont acceptées
    xstream.addPermission(AnyTypePermission.ANY);
    return xstream;
}
```

Les blocs notés {OUT} et {CFG} servent respectivement à configurer la production du XML en sortie et à configurer les inspections. Voir UtilXML.java pour le détail, parce que c'est très complexe, voir xstream issue 101.

Ajustement de la sortie XML

XStream donne des noms basés sur les noms des classes aux éléments. Cela peut être déplaisant. Voici plusieurs possibilités de configuration :

- changer le nom de l'élément représentant une classe : `xstream.alias("NOM", CLASSE.class);`
- mettre une variable membre en attribut : `xstream.useAttributeFor(CLASSE.class, "MEMBRE");`
- mettre une collection à plat au lieu de l'imbriquer dans un élément racine : `xstream.addImplicitCollection(CLASSE.class, "MEMBRE");`

Ajustement de la sortie XML, suite

Par exemple, pour les voitures possédées par un client :

```
class Client {
    private int id;
    private List<Voiture> voitures;
}
```

```
    xstream.useAttributeFor(Client.class, "id");
    xstream.alias("client", Client.class);
    xstream.alias("voiture", Voiture.class);
    xstream.addImplicitCollection(Client.class, "voitures");
```

```
<client id="1">
    <voiture id="1">...</voiture>
    <voiture id="3">...</voiture>
    <voiture id="7">...</voiture>
</client>
```

Sérialisation avec l'API Simple

Présentation

Simple est une bibliothèque ressemblant à JAXB (devenue obsolète) pour la sérialisation XML d'objets Java. Elle est extrêmement simple à utiliser. Il suffit d'appliquer des *annotations* sur des classes pour les rendre automatiquement sérialisables.

```
import org.simpleframework.xml.*;

@Root class Voiture {
    @Attribute   private int id;
    @Element     private String modele;
    @Element     private float prix;
}
@Root class Client {
    @Attribute   private int id;
    @ElementList private List<Voiture> voitures;
}
```

Annotations

Une annotation Java, voir wikipedia est une sorte de fonction qui s'applique au code source. Le résultat peut être une simple vérification comme `@Override`, ou une directive de compilation `@SuppressWarnings`, ou des propriétés de variables `@Nullable`.

Dans le cas de Simple, quelques annotations suffisent pour générer les instructions permettant de sérialiser la classe :

- `@Root` à placer avant le nom de la classe, pour générer un élément XML du même nom que la classe.
- `@Attribute` à placer devant une variable membre qu'il faut écrire sous forme d'attribut.
- `@Element`, à placer devant une variable membre qu'il faut écrire sous forme d'élément.
- `@ElementList`, à placer devant une variable membre de type collection.

Exemple

Voici un exemple d'annotations placées sur une classe :

```
@Root class Voiture {
    @Attribute private int id;
    @Element   private String modele;
    @Element   private float kilometrage;
}
```

La sérialisation d'une instance produira :

```
<voiture id="321">
    <modele>Bugatti Chiron</modele>
    <kilometrage>12499</kilometrage>
</voiture>
```

Sérialisation

La méthode toXML s'écrit alors :

```
import org.simpleframework.xml.*;
import org.simpleframework.xml.core.*;
import java.io.File;

public void toXML(String filename) throws Exception
{
    Serializer serializer = new Persister();
    File output = new File(filename);
    serializer.write(this, output);
}
```

Pour la lancer :

```
    maVoiture.toXML("voiture.xml");
```


Pierre Nerzic

Désérialisation

La méthode `fromXML` s'écrit :

```
public static Voiture fromXML(String filename) throws Exception
{
    Serializer serializer = new Persister();
    File input = new File(filename);
    return serializer.read(Voiture.class, input);
}
```

Pour la lancer :

```
    Voiture maVoiture = Voiture.fromXML("voiture.xml");
```


Pierre Nerzic

Ajustement de la sortie XML

Dans le cas des variables membres de type collections, on peut demander à Simple de ne pas produire un élément englobant. Il suffit de paramétrer l'annotation `@ElementList` avec `inline=true` :

```
@Root class Client {
    @Attribute   private int id;
    @Element     private String adresse;

    @ElementList(inline=true) private List<Voiture> voitures;
}
```

Avec ces annotations, les différents éléments de la liste seront mis directement en dessous de l'élément représentant le client.

API DOM dans d'autres langages

Pierre Nerzic

Résumé

L'API W3C DOM existe pour de nombreux langages de programmation : JavaScript, PHP, Python, etc. Elle est d'emploi quasiment identique. Parfois, comme en Python et JavaScript, de nombreuses fonctions comme `document.getDocumentElement()`, `element.getChildNodes()` sont remplacées par des accès directs aux propriétés : `document.documentElement`, `element.childNodes`.

Consulter par exemple les documentations de la classe `Node` :

- en Java,
- en JavaScript,
- en Python.

Création d'un document XML en JavaScript

Pour illustrer l'API en JavaScript, voici d'abord la création d'un document XML, et pour commencer, le cadre général :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <script type="text/javascript">
            <!-- FONCTIONS JAVASCRIPT ICI -->
        </script>
    </head>
    <body onload="main()">
        <p>Tapez CTRL U pour voir le source...</p>
        <pre id="affichage"></pre>
    </body>
</html>
```


Pierre Nerzic

Script de création d'un document

Voici tout d'abord la création du document XML avec une racine appelée "voitures" :

```
function main() {
    let URI = "";
    let nomracine = "voitures";
    let Doctype = null;
    let XMLdoc = document.implementation.createDocument(
                    URI, nomracine, Doctype);
    let racine = XMLdoc.documentElement;
```

On peut fournir un URI pour placer tous les éléments dans un *namespace*. Il faut alors mettre le même préfixe à tous les éléments de cet URI et les créer avec `createElement(URI,"`*préfixe*`:nom")`

NB: la variable ne peut pas s'appeler `document` car c'est le nom du document HTML dans le navigateur.

Création d'éléments

L'ajout d'éléments, d'attributs et de textes ressemble à ce qu'on fait en Java :

```
let voiture1 = XMLdoc.createElement("voiture");
voiture1.setAttribute("marque", "Renault");
racine.appendChild(voiture1);

let voiture2 = XMLdoc.createElement("voiture");
voiture2.appendChild(XMLdoc.createTextNode("Peugeot"));
racine.appendChild(voiture2);
```

Affichage du résultat

Pour finir, le résultat peut être affiché dans le document HTML par un *serializer* :

```
    let serializer = new XMLSerializer();
    let xml = serializer.serializeToString(XMLdoc);
    xml = xml.replace(/&/g, "&amp;");
    xml = xml.replace(/</g, "&lt;");
    xml = xml.replace(/>/g, "&gt;");
    xml = xml.replace(/\"/g, "&quot;");
    xml = xml.replace(/\'/g, "&apos;");
    document.getElementById("affichage").innerHTML = xml;
}
```

Notez le remplacement de certains caractères par les entités HTML.

Pierre Nerzic

Parcours d'un fichier XML

On en arrive au plus utile dans un client HTTP, l'utilisation de données reçues du réseau, en général par AJAX.

```
function main() {
    var requete = new XMLHttpRequest();
    requete.onreadystatechange = function() {
        if (requete.readyState == 4 && requete.status == 200) {
            TraiterReponse(requete.responseXML);
        }
    }
    requete.open("GET", "voitures.xml", true);
    requete.send();
}
```

La demande de téléchargement et la réponse du serveur sont asynchrones. Lorsque le fichier arrive, ça appelle `TraiterReponse`.

Pierre Nerzic

Traitement de la réponse HTTP

Par exemple, on compte les éléments <voiture> (méthode 1) :

```
function TraiterReponse(document) {
    let racine = document.documentElement;
    let nbvoitures = 0;
    let courant = racine.firstChild;
    while (courant != null) {
        if (courant.nodeType == Node.ELEMENT_NODE &&
            courant.nodeName == "voiture") {
            nbvoitures = nbvoitures + 1;
        }
        courant = courant.nextSibling;
    }
    document.getElementById("affichage").innerHTML =
            nbvoitures+" voitures";
}
```


Pierre Nerzic

Validation en JAVA

Pierre Nerzic

Présentation

L'API Java `javax.xml.validation` fournit tout ce qui permet de valider un document XML contre une DTD ou un Schéma.

Pour valider par une DTD, c'est très simple, il faut qu'il mentionne sa DTD dans une balise `<!DOCTYPE>` et il suffit de l'ouvrir ainsi :

```
// créer un constructeur avec validation
DocumentBuilderFactory factory = DocumentBuilderFactory.newInsta
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
// lire le fichier pour remplir le document
Document document = builder.parse(new File("document.xml"));
```

Toute exception indique qu'il n'est pas valide.

Validation par un schéma

Par exemple, pour valider `document.xml` par `document.xsd` :

```
// lire le document xml
DocumentBuilder builder = DocumentBuilderFactory.newInstance().n
Document document = builder.parse(new File("document.xml"));
// créer un validateur basé sur le schéma
SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W
Source schemaFile = new StreamSource(new File("document.xsd"));
Schema schema = factory.newSchema(schemaFile);
Validator validator = schema.newValidator();
// valider le document par le schéma
try {
    validator.validate(new DOMSource(document));
} catch (SAXException e) {
    // document non valide
}
```


Pierre Nerzic