

Introduction à Stax

par Gildas Cuisinier ([Hikage](#)) ([Blog](#))

Date de publication : 19/10/2007

Dernière mise à jour :

A l'heure où SAX et DOM n'ont plus à faire leur preuve dans le traitement de document XML, une nouvelle API fait son apparition dans le monde Java. Celle-ci reprend les avantages de SAX, sa rapidité et faible utilisation, tout en fournissant un moyen de créer des documents XML à l'instar de DOM. Cet article a pour but de vous présenter les bases de l'utilisation de cette nouvelle API : StAX.
Bonne lecture !

- I - Introduction
- II - Stax
 - II-A - Projets utilisant StAx
 - II-B - Comment trouver Stax ?
- III - Lecture d'un document avec StAx
 - III-A - Sélection de l'implémentation
 - III-B - XMLInputFactory
 - III-C - XmlStreamReader
 - III-D - Exemple de lecture
- IV - Ecriture d'un document avec StAx
 - IV-A - Sélection d'une implémentation
 - IV-B - XMLOutputFactory
 - IV-C - XMLStreamWriter
 - IV-C-1 - Création d'un élément
 - IV-C-2 - Ajout d'un attribut à un élément
 - IV-C-3 - Ajout de texte
 - IV-C-4 - Ajout de commentaire
 - IV-D - Conclusion sur l'écriture
- V - API Événementielle & Filtres
 - V-A - XMLEventReader
 - V-B - XMLEventWriter
- VI - Conclusion
 - VI-A - Remerciements

I - Introduction

A l'heure actuelle, le langage XML est de plus en plus présent dans le monde informatique : présentation web avec l'XHTML, documents avec l'Open Document et l'OpenXML, image vectorielle avec SVG ou encore services web avec SOAP/WSDL.

Pour traiter ce format, il existe deux méthodes bien connues : SAX et DOM.

De son côté, SAX (ou Simple API for XML) est de type événementiel. Sans rentrer dans le détail, un parseur SAX va lire un document XML et va considérer l'ouverture d'une balise, sa fermeture, le début d'une zone de texte comme des événements.

Et pour chacun de ceux-ci correspond une méthode qui sera appelée par le parseur. C'est l'ensemble de ces méthodes qui devront être fournies par le développeur afin de traiter le document XML.

L'avantage de cette méthode est qu'elle est très rapide et très peu gourmande en mémoire. En contre partie, le traitement est fait en une seule passe, et ne fournit pas d'accès aléatoire aux données du fichier.

De l'autre côté, DOM fonctionne différemment : un parseur DOM va lire un document XML et va créer la représentation mémoire de celui-ci sous forme d'un arbre. Cette représentation implique une forte utilisation mémoire ainsi qu'un temps de traitement plus long que SAX. Mais DOM a pour avantage de fournir un accès aléatoire et illimité aux données.

De plus, DOM n'est pas limité à la lecture d'un fichier XML. En effet, une fois un arbre créé en mémoire, il est tout à fait possible de le modifier (supprimer, ajouter, modifier ses noeuds), voir même d'en créer un nouveau, pour ensuite l'exporter vers un flux.

II - Stax

Au vue du succès de XML, une nouvelle API est apparue dans le monde Java : StAx

Celle-ci possède des qualités empruntées à la fois à SAX et à DOM :

- Elle est peu gourmande en mémoire
- Elle permet l'écriture de fichier XML
- Elle fournit un accès séquentiel aux données uniquement (pas de retour en arrière dans le document)

Cependant, malgré ces ressemblances, Stax diffère tout de même des deux autres.

En effet, dans SAX, le traitement est dirigé par le parseur SAX. C'est lui qui va lire le document et générer les évènements au fur et à mesure qu'il progresse dans le document XML. Le code du développeur a un rôle passif : il traite les évènements du parseur mais ne possède aucun contrôle sur l'avancement dans le document.

Dans StAx, c'est au contraire le code du développeur qui va faire les demandes au parseur, afin que celui-ci fournisse l'élément suivant dans le document.

La fonctionnalité d'écriture de StAx diffère aussi de celle de DOM. DOM a besoin que l'arbre soit complètement construit avant de pouvoir l'exporter sur un flux. Ce qui peut parfois prendre énormément de temps.

De son côté, StAx permet d'envoyer sur un flux les éléments directement à leur création, ce qui a comme contrainte qu'il faut s'assurer que la construction sera séquentielle, et qu'aucune balise ne sera ajoutée dans une section déjà envoyée.

Cette contrainte mise à part, l'optimisation mémoire est importante, spécialement lors de l'écriture de gros document XML.

De plus, dans un contexte client/serveur (service web par exemple), il y a aussi une optimisation temporelle. Dans le cas de DOM, le client doit attendre que le serveur crée l'arbre DOM, puis doit attendre la réception complète du document XML pour seulement commencer le traitement.

Avec StAx, le client reçoit les données directement, et peut donc travailler simultanément avec le serveur.

En plus de fournir des fonctionnalités de lecture et d'écriture, StAx est composée de deux API.

La première est l'API Curseur : c'est une API de bas niveau, dont le fonctionnement se base sur un curseur virtuel qui permet de connaître la position dans le document XML et ainsi autorise l'utilisation de certaines méthodes pour récupérer les informations.

La seconde est dite événementielle. Elle est de plus haut niveau, l'API va créer des objets représentant l'élément sur lequel elle se situe. Elle est un peu plus gourmande en mémoire (mais nettement moins que DOM) et un peu plus lente à cause de la création de ces objets.

II-A - Projets utilisant StAx

StAx bien que récente par rapport au deux autres, on compte déjà quelques projets l'utilisant. Voici une liste non exhaustive de projet l'utilisant :

Nom de l'API	Description
XFire / CXF	Framework pour l'implémentation de services Web en Java
Axiom	Axis Object Model, le coeur de Axis 2 (Web Service)
Apache Scout	Implémentation de l'API XML Registries (JAXR)

On remarque que StAx est fort présent dans tout ce qui touche au Web Services, rien d'étonnant lorsqu'on sait que ceux-ci se basent sur des messages SOAP, qui eux-même sont des documents XML.

II-B - Comment trouver Stax ?

Comme beaucoup d'API en Java, StAx est définie via des interfaces, et il existe plusieurs implémentations disponibles dont

Nom	Description	Url	Licence
RI	Implémentation de référence. Développée au départ par Bea, et mise à disposition de l'Open Source via CodeHaus.org	http://stax.codehaus.org	Apache ASL 2.1
SJSXP	Implémentation de Sun Microsystem. Fournie par défaut dans Java 6	https://sjsxp.dev.java.net/	/
Woodstox	Implémentation complète de Stax (y compris les parties optionnelles). Celle-ci propose des extensions qui pourraient à terme devenir un StAx2	http://woodstox.codehaus.org	LGPL 2.1 / ASL 2.0.

Afin de pouvoir utiliser StAx, il est donc nécessaire de posséder l'API et une de ses implémentations. Comme le fait remarquer le tableau ci-dessus, l'implémentation StAx de Sun Microsystem ainsi que l'API sont fournies dans la JRE 6.

Dans le cas où Maven 2 est utilisé pour la gestion de votre projet, il suffit d'ajouter la dépendance de l'implémentation, par exemple dans le cas de l'implémentation de référence :

```
<dependency>
  <groupId>stax</groupId>
  <artifactId>stax</artifactId>
  <version>1.2.0</version>
</dependency>
```

III - Lecture d'un document avec StAx

III-A - Sélection de l'implémentation

La première étape dans l'utilisation de StAx est de récupérer une implémentation sur un des sites précités.

Mais comme il est tout à fait possible de posséder plusieurs implémentations de StAx dans le classpath, l'API possède un système pour spécifier laquelle utiliser, qui dans l'ordre est :

- 1 Si une propriété système **javax.xml.stream.XmlInputFactory** est définie (avec le nom d'une implémentation), alors c'est celle-ci qui sera utilisée.
- 2 Stax va lire le fichier **xml.stream.properties** situé dans le répertoire **lib** du JRE s'il est présent
- 3 Stax va chercher après un fichier **META-INF/services/javax.xml.stream**
- 4 StAx va utiliser l'implémentation par défaut si elle est disponible

III-B - XMLInputFactory

Une fois l'implémentation choisie, il est possible de récupérer une instance de fabrique de parseur :

```
XMLInputFactory factory = XMLInputFactory.newInstance();
```

C'est grâce aux méthodes de cette instance qu'il sera ensuite possible de récupérer un parseur, de type Curseur ou événementiel.

Mais le rôle de la fabrique n'est pas restreint à la création d'un de ceux-ci mais aussi de les configurer par le biais des propriétés.

Il existe deux types de propriétés : les propriétés booléennes et les propriétés objet.

Les premières permettent d'activer ou désactiver des fonctionnalités telles que la validation, la gestion des schémas ou encore l'acceptation de DTD. Par exemple, si l'on désire activer la validation du document XML :

```
factory.setProperties("javax.xml.stream.isValidating", Boolean.TRUE);
```

Mais certaines propriétés ne sont pas définies comme obligatoires dans l'API, il est donc nécessaire de vérifier cela au préalable :

```
if(factory.isPropertySupported("javax.xml.stream.isValidating"))  
factory.setProperties("javax.xml.stream.isValidating", Boolean.TRUE);
```

L'exemple le plus fréquent au niveau de propriétés de type objet, est l'XMLReporter. En effet par défaut, StAx générera des exceptions en cas d'erreur fatale (c-à-d conduisant à l'impossibilité de continuer le traitement) mais n'informerait pas d'erreur moins graves ou même des alertes.

Le fait de spécifier un XMLReporter permet de récupérer celles-ci.

```
factory.setXMLReporter(new XMLReporter() {
```

```

        public void report(String message, String typeErreur, Object source, Location location)
        throws XMLStreamException {
            System.out.println("Erreur de type : " + typeErreur + ", message : " + message);
        }
    };
    
```

Dès lors, les erreurs et alertes seront reportées via l'XMLReporter.

III-C - XmlStreamReader

Une fois que la fabrique est configurée, il est possible de récupérer un parseur via les méthodes `createXMLStreamReader(*)` et `createXMLEventReader(*)`. Ces méthodes permettent de créer respectivement un parseur de type curseur ou de type événementiel.

Chaque méthode est surchargée, afin de prendre diverses sources (un flux binaire, un flux de caractères, ou une source de l'API TrAx) en entrée.

Afin de créer un parseur de type curseur sur un fichier xml situé sur le disque, le code suivant pourrait être utilisé :

```

File file = new File("sample.xml");

XMLStreamReader reader = factory.createXMLStreamReader(new FileReader(file));
    
```

Une fois fait, l'itération se fait via les méthodes `hasNext()` et `next()` :

```

while (reader.hasNext()) {
    int type = reader.next();

    // traitements
}
    
```

La première vérifie qu'il y a bien un élément à traiter (`hasNext()`), la seconde (`next()`) avance le curseur sur celui-ci et renvoie son type sous forme d'un entier. Cet entier est une valeur parmi celle de ce tableau :

Constante	Valeur	Description
START_ELEMENT	1	Début d'une balise
END_ELEMENT	2	Fin d'une balise
PROCESSING_INSTRUCTION	3	Instruction de traitements
CHARACTERS	4	Chaine de caractères
COMMENT	5	Commentaire XML
SPACE	6	Espace blanc
START_DOCUMENT	7	Début du document
END_DOCUMENT	8	Fin du document XML
ENTITY_REFERENCE	9	Référence d'entité (exemple :)
ATTRIBUTE	10	Attribut d'une balise
DTD	11	Déclaration d'une DTD
CDATA	12	Zone CDATA
NAMESPACE	13	Déclaration d'un schéma XML
NOTATION_DECLARATION	14	

ENTITY_DECLARATION	15	
--------------------	----	--

Le nombre est assez important, mais seules quelques unes sont généralement utilisées.

L'importance de connaître le type de l'élément est qu'il va définir quelles méthodes du parseur seront réellement accessibles. Celui-ci possède effectivement un certain nombre de méthodes dont le but est de récupérer le nom de la balise, ses attributs, le texte qu'il contient, ...

Dans le cas où une méthode du parseur est appelée, mais que le type de l'élément courant ne la permet pas, une exception de type **IllegalStateException** est lancée.

III-D - Exemple de lecture

Voici un exemple simple de traitement d'un fichier XML contenant des utilisateurs et leur mot de passe associés :

```
List<User> users = new ArrayList<User>();
while (reader.hasNext()) {
    // Récupération du type de l'élément
    int type = reader.next();

    switch (type) {
        case XMLStreamReader.START_ELEMENT:
            // Si c'est un début de balise user, alors on lance le traitement d'un utilisateur
            if ("user".equals(reader.getLocalName()))
                processUser(users, reader);
            break;
    }
}
```

```
private static void processUser(List<User> users, XMLStreamReader reader) throws XMLStreamException
{
    int flag = 0;
    final int FLAG_NONE = 0;
    final int FLAG_USERNAME = 1;
    final int FLAG_PASSWORD = 2;
    final int FLAG_DESCRIPTION = 3;

    boolean state = true;
    while (reader.hasNext() && state) {
        int type = reader.next();

        switch (type) {
            // Si c'est un début d'élément, on garde son type
            case XMLStreamReader.START_ELEMENT:
                if (reader.getLocalName().equals("login"))
                    flag = FLAG_USERNAME;
                else if (reader.getLocalName().equals("password"))
                    flag = FLAG_PASSWORD;
                else if (reader.getLocalName().equals("description"))
                    flag = FLAG_DESCRIPTION;
                else flag = FLAG_NONE;
                break;

            // Si c'est du texte ou une zone CDATA ...
            case XMLStreamReader.CDATA:
            case XMLStreamReader.CHARACTERS:
                switch (flag) {
                    case FLAG_DESCRIPTION:

```

```
        // et que ce n'est pas une chaîne de blancs
        if(!(reader.isWhiteSpace()))
            UserBuilder.setDescription(reader.getText());
        break;
    case FLAG_PASSWORD:
        // et que ce n'est pas une chaîne de blanc
        if(!(reader.isWhiteSpace()))
            UserBuilder.setPassword(reader.getText());
        break;
    case FLAG_USERNAME:
        // et que ce n'est pas une chaîne de blanc
        if(!(reader.isWhiteSpace()))
            UserBuilder.setLogin(reader.getText());
        break;
    }
    break;
case XMLStreamReader.END_ELEMENT:
    // Si c'est la fin de la balise user, on change l'indicateur de boucle
    if (reader.getLocalName().equals("user")) state = false;
    break;
}
}
users.add(UserBuilder.createUser());
}
```

Voici qui conclut la partie sur la lecture avec l'API Curseur de StAx.

IV - Ecriture d'un document avec StAx

Avant de présenter l'autre API de lecture (orientée événements), il est intéressant de voir comment il est possible de créer un document XML avec StAx.

IV-A - Sélection d'une implémentation

Tout comme pour la lecture, il est possible de spécifier quelle implémentation sera utilisée. Le principe est exactement le même, sauf que c'est une propriété `javax.xml.stream.XmlOutputFactory` qui sera recherchée.

IV-B - XMLOutputFactory

Vous l'aurez deviné, une fabrique est nécessaire afin de récupérer une instance d'un "générateur xml" StAx.

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();  
writer = outputFactory.createXMLStreamWriter(System.out);
```

IV-C - XMLStreamWriter

Une fois l'instance d'`XMLStreamWriter` récupérée, c'est grâce aux diverses méthodes `writeXXXX` que le document sera écrit.

La première étant toujours `writeStartDocument`, qui prend 0, 1 ou 2 paramètres de type **String**. Le premier paramètre étant la version XML du document, le second étant la spécification du thème d'encodage.

Si aucun paramètre n'est fourni, les valeurs par défaut sont 1.0 et UTF-8.

```
writer.writeStartDocument();  
// donnera <?xml version='1.0' encoding='utf-8'?>
```

IV-C-1 - Création d'un élément

Afin de créer un élément, ce sont les méthodes `writeStartElement()` (et `writeEndElement()`) qui seront utilisées.

Les méthodes `writeStartElement` prennent obligatoirement en paramètre le nom de la balise à créer, et optionnellement le namespace et son préfixe.

```
void writeStartElement(java.lang.String nomElement) throws javax.xml.stream.XMLStreamException;  
void writeStartElement(java.lang.String namespaceUri, java.lang.String nomElement) throws  
    javax.xml.stream.XMLStreamException;  
void writeStartElement(java.lang.String string, java.lang.String string1, java.lang.String  
    nameSpaceUri) throws javax.xml.stream.XMLStreamException;
```

Un petit exemple :

```
// Début du document
```

```
writer.writeStartDocument();

// Ecriture de l'élément conteneur
writer.writeStartElement( "dvp", "element", "http://hikage.developpez.com/stax");
// Ajout d'un namespace
writer.writeNamespace( "dvp", "http://hikage.developpez.com/stax");


// Ecriture d'un élément fils
writer.writeStartElement( "http://hikage.developpez.com/stax", "subElement");
//Fermeture de l'élément fils
writer.writeEndElement();

// Fermeture de l'élément conteneur
writer.writeEndElement();

// Fin du document
writer.writeEndDocument();
writer.close();
```

Donnera :

```
<?xml version='1.0' encoding='utf-8'?><dvp:element
  xmlns:dvp="http://hikage.developpez.com/stax"><dvp:subElement></dvp:subElement></dvp:element>
```

 *A titre d'information, cette sortie est tout à fait normale. En effet, StAx ne fournit pas en standard un système de formatage du document XML.*

Le projet stax-utils ([site du projet](#)) fournit un Writer qui génère du code formaté.

On peut voir dans cette petite portion de code différents éléments. Premièrement, la création d'un élément **element** avec la déclaration du namespace "dvp". Ensuite, la création d'une élément **subElement** comme fils du premier élément.

IV-C-2 - Ajout d'un attribut à un élément

Maintenant que la création de balises est expliquée, il serait intéressant de pouvoir leur ajouter des attributs.

Pour cela, ce sont les méthodes `writeAttribute()`. Les paramètres obligatoires sont le nom de l'attribut ainsi que sa valeur. Tout comme pour les éléments, il est possible de spécifier leur namespace.

```
writer.writeStartElement( "http://hikage.developpez.com/stax", "subElement");
//Attribut avec spécification du namespace
writer.writeAttribute( "http://hikage.developpez.com/stax", "attribut1", "valeur1");
//Attribut sans namespace
writer.writeAttribute( "attribut2", "valeur2");
//Fermeture de l'element fils
writer.writeEndElement();
```

Il est intéressant de voir qu'il n'y a pas de lien direct entre un élément et une balise. Le fait d'écrire un attribut l'associe à la dernière balise ouverte, si cela est possible.

Par exemple, le code suivant produira une erreur :

```
writer.writeStartElement( "http://hikage.developpez.com/stax", "subElement" );
// Ajout d'une zone CDATA
writer.writeCdata( "test" );
//Attribut avec spécification du namespace, après fermeture > de la balise
writer.writeAttribute( "http://hikage.developpez.com/stax", "attribut1", "valeur1" );
//Fermeture de l'élément fils
writer.writeEndElement();
```

La cause de cela, est que le fait d'appeler une méthode pour écrire une zone CDATA (ou autre), va obliger StAx à finir le StartElement et donc rendre impossible l'ajout d'attribut.

IV-C-3 - Ajout de texte

Un autre type d'information contenu dans une balise sont des éléments textuels. Pour cela, ce sont les méthodes *writeCharacters()* qui interviennent :

```
writer.writeStartElement( "http://hikage.developpez.com/stax", "subElement" );

writer.writeCharacters("Ceci est un texte");

writer.writeEndElement();
```

De plus, si le texte contient des caractères spéciaux tels que <, >, &, ceux-ci seront remplacés par leur représentation sous forme d'entité : <, >, &

Dans le cas où le texte doit absolument être écrit tel quel, la méthode *writeCdata()* sera plus appropriée : elle englobera le texte dans une zone CDATA sans modification.

IV-C-4 - Ajout de commentaire

Il est tout à fait possible d'ajouter des commentaires XML avec l'API StAx, et ce via la méthode *writeComment()*

```
writer.writeComment( "Ceci est un commentaire" );
```

IV-D - Conclusion sur l'écriture

Lorsqu'on décide d'utiliser StAx pour la création de document XML, il est important de prendre en compte que l'API ne garantit pas la création d'un document bien formé ni même valide par rapport à un schéma ou autre.

La seule garantie qu'offre StAx est que l'appel de *writeEndDocument()* fermera toutes balises non fermées :

```
writer.writeStartDocument();

writer.writeStartElement( "element1" );
writer.writeStartElement( "element2" );
writer.writeStartElement( "element3" );
writer.writeStartElement( "element4" );

//Il manque la fin du document

writer.close();
```

```
<?xml version='1.0' encoding='utf-8'?><element1><element2><element3><element4
```

On remarque que les ouvertures des éléments sont correctement réalisées pour les trois premières, mais pas pour la dernière.

La cause de cela, c'est que la méthode *writeEndElement()* n'est pas appelée explicitement et qu'aucun autre élément n'oblige StAx à fermer la dernière balise.

De plus, aucune balise de fin n'est présente dans le résultat.

Le même code, appelant *writeEndDocument()* cette fois ci :

```
// Début du document
writer.writeStartDocument();

writer.writeStartElement("element1");
writer.writeStartElement("element2");
writer.writeStartElement("element3");
writer.writeStartElement("element4");

writer.writeEndDocument();

writer.close();
```

donnera une sortie complètement différente :

```
<?xml version='1.0'
encoding='utf-8'?><element1><element2><element3><element4></element4></element3></element2></
element1>
```

V - API Événementielle & Filtres

V-A - XMLEventReader

Comme il l'a été dit, StAx est en réalité constitué de deux API, la première dite curseur et la seconde dite événementielle.

L'utilisation de cette dernière ne diffère pas énormément de la première.

Elle nécessite l'utilisation d'une fabrique pour instancier le parseur, le parseur quant à lui contient aussi les méthodes `hasNext()` et `next()`.

La différence se situe dans le type de retour de la méthode `next()`. Dans le cas de l'API curseur, c'était un entier qui définissait le type d'élément, et par conséquent les méthodes possibles dans le parseur lui même.

Dans le cas de l'Event API, c'est un objet dérivant de `XMLEvent` qui est retourné :

```
XMLEventReader reader = factory.createXMLEventReader(new FileReader(url));

while(reader.hasNext()){
    XMLEvent next = reader.nextEvent();
}
```

`XMLEvent` ne fournit pas énormément de méthodes, si ce n'est des méthodes pour récupérer le type exact de l'élément, afin de pouvoir caster correctement l'objet par la suite :

```
if(next.isStartElement()){
    StartElement start = next.asStartElement();
}
```

Une fois que le véritable type de l'`XMLEvent` est récupéré, les méthodes d'accès aux données sont utilisables. Dans le cas d'un `StartElement`, les méthodes de récupération des attributs, du nom de l'élément ainsi que de l'espace de nom sont disponibles. Dans le cas d'un commentaire, la seule méthode supplémentaire est celle pour récupérer le texte de celui-ci.

L'avantage est donc qu'il est impossible d'appeler une méthode qui n'est pas utilisable sur l'élément courant, vu qu'une méthode non utilisable n'est pas définie dans l'interface du type de l'élément courant.

Un autre avantage est qu'une fois l'`XMLEvent` instancié, celui-ci est immuable, et donc, même si le parseur progresse dans le document XML, les informations seront toujours disponibles.

Mais le plus grand atout de l'Event API est l'utilisation d'objet récupéré en lecture, pour écrire avec l'API de création.

V-B - XMLEventWriter

Une fois de plus, l'utilisation de `Writer` de l'Event API ressemble beaucoup à celle de l'API Curseur :

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLEventWriter writer = outputFactory.createXMLEventWriter(System.out);
```

La création des divers éléments passe par une fabrique d'événements :

```
XMLEventFactory eventFactory = XMLEventFactory.newInstance();
```

Celle-ci met à disposition des méthodes pour créer tous les types d'événement (StartElement, StartDocument, ...).

Les XMLEvent ainsi créés sont ensuite simplement passés comme paramètre à la méthode `add()` de l'`XMLEventWriter` :

```
XMLEvent tmp = eventFactory.createComment("Test de commentaire");  
  
writer.add(tmp);
```

Mais la force de StAx est l'utilisation des mêmes interfaces XMLEvent pour la lecture et l'écriture. Il est tout à fait correct de copier un document XML de cette façon :

```
File url = new File("sample.xml");  
  
XMLEventReader reader = factory.createXMLEventReader(new FileReader(url));  
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();  
XMLEventWriter writer = outputFactory.createXMLEventWriter(System.out);  
  
while(reader.hasNext()){  
    writer.add(reader.nextEvent());  
}  
  
writer.close();
```

Bon, bien sûr, il n'y a rien de très intéressant en soi à prendre un fichier XML pour l'envoyer vers un autre flux.

Imaginons maintenant, que pour une raison ou une autre, on désirerait ne récupérer qu'une partie d'un document :

```
<document xmlns:meta="http://hikage.developpez.com/stax">  
  <meta:auteur>  
    <meta:nom>Cuisinier</meta:nom>  
    <meta:prenom>Gildas</meta:prenom>  
    <meta:website>http://www.developpez.com</meta:website>  
    <meta:email>gildas.cuisinier at redaction-developpez.com</meta:email>  
  </meta:auteur>  
  
  <article>  
    <!-- Le reste du document ici -->  
  </article>  
  
</document>
```

Dans ce document XML, nous désirons récupérer uniquement les métadonnées auteurs (qui sont préfixées par l'espace de nom `http://hikage.developpez.com/stax`).

Pour cela, StAx fournit des filtres : `EventFilter` pour l'API événementielle, et `StreamFilter` pour l'API Curseur.

Dans notre cas, le filtre ressemblerait à ceci :

```
EventFilter filter = new EventFilter() {
    boolean headerData = false;

    public boolean accept(XMLEvent xmlEvent) {

        if (xmlEvent.isStartElement()) {
            StartElement start = xmlEvent.asStartElement();

            // si c'est le début du tag auteur
            if (start.getName().getNamespaceURI().equals("http://hikage.developpez.com/stax") &&
start.getName().getLocalPart().equals("auteur")) {
                headerData = true;
                return true;
            }
        } else if (xmlEvent.isEndElement()) {
            EndElement end = xmlEvent.asEndElement();
            // si c'est la fin du tag auteur
            if (end.getName().getNamespaceURI().equals("http://hikage.developpez.com/stax") &&
end.getName().getLocalPart().equals("auteur")) {
                headerData = false;
                return true;
            }
        }

        return headerData;
    }
};
```

Une fois l'instance de ce filtre créée, il faut l'associer à un `XMLEventReader` via la méthode `createFilteredReader()` de la fabrique `XMLInputFactory` :

```
filteredReader = factory.createFilteredReader(reader, filter);
```

Il ne reste plus qu'à utiliser ce `Reader` à la place du premier, ce qui devrait donner la sortie suivante :

```
<meta:auteur>
  <meta:nom>Cuisinier</meta:nom>
  <meta:prenom>Gildas</meta:prenom>
  <meta:website>http://www.developpez.com</meta:website>
  <meta:email>gildas.cuisinier at redaction-developpez.com</meta:email>
</meta:auteur>
```

VI - Conclusion

Voilà qui termine cette brève introduction de StAx, qui n'a pas la prétention de montrer toutes les possibilités de StAx mais qui devrait suffire à vous permettre de mieux comprendre et surtout d'utiliser StAx !

VI-A - Remerciements

Je tiens à remercier Julien Vasseur ainsi que **julp**, **Vincent Brabant** et **Ricky81** pour leur relecture technique et orthographique.

