47. StAX (Streaming Api for XML)

Niveau : Intermédiaire

StAX est l'acronyme de Streaming Api for XML : c'est une API qui permet de traiter un document XML de façon simple en consommant peu de mémoire tout en permettant de garder le contrôle sur les opérations d'analyse ou d'écriture.

StAX a été développée sous la JSR-173 et est incorporée dans Java SE 6.0.

StAX propose des fonctionnalités pour parcourir et écrire un document XML mais ne permet pas de manipuler le contenu d'un document.

Le but de StAX n'est pas de remplacer SAX ou DOM mais de proposer une nouvelle façon d'analyser un document XML : StAX vient en complément des API DOM et SAX.

Sa mise en oeuvre par rapport aux deux API existantes peut être dans certains cas plus simple et donc plus facile que SAX et plus efficace et performante que DOM. StAX permet de traiter un document XML de manière rapide, facile et consommant peu de ressources : le modèle d'événements utilisé est plus simple que celui de SAX et les ressources requises sont moins importantes que pour un traitement grâce à DOM.

Ce chapitre contient plusieurs sections :

47.1. La présentation de StAX

Avant StAX, l'analyse d'un document XML pouvait se faire principalement par deux API standard (DOM et SAX) ou une API non standard (JDOM).

L'analyse d'un document XML peut se faire grâce à deux grandes catégories de parseurs :

- Parseur basé sur un arbre : DOM implémente cette technique qui consiste à représenter et stocker le document dans un arbre d'objets. Il est ainsi possible, une fois cet arbre créé, de parcourir librement les noeuds de l'arbre et de le modifier. Cette représentation est généralement plus gourmande en ressource que le document lui-même ce qui la rend particulièrement inadaptée à des documents de grandes tailles.
- Parseur basé sur un flux (document streaming) : des événements sont émis lors de la lecture séquentielle du flux pour notifier chaque changement lors de l'analyse du document. SAX implémente cette technique qui nécessite moins de ressources mais offre cependant moins de souplesse dans la manipulation du document.

Il existe deux sortes de traitement par flux :

- Push : le parser génère des événements à chaque noeud du document que le client en ait besoin ou non
- Pull : le client demande explicitement au parser de lui donner l'événement suivant ce qui permet au client de conserver la main sur les traitements du parser et ainsi de piloter l'analyse

	Push parsing	Pull parsing
Implémentation	SAX	StAX
Contrôle des traitements	Par le parseur	Par le client
Complexité de mise en oeuvre	Moyenne	Faible
Parcours de tout le document	Oui	Non (interruption possible par le client)

Avec le traitement par flux, seul l'élément courant durant le parcours séquentiel du document est accessible. Ceci limite les traitements possibles sur le document et impose généralement de conserver un contexte.

L'utilisation d'un traitement par flux est particulièrement utile lors de la manipulation de gros documents, de l'utilisation dans un environnement possédant des ressources limitées (exemple utilisation avec Java ME) ou lors de traitements en parallèle de documents (exemple dans un serveur d'applications ou un moteur de services web).

StAX propose un modèle de traitement du document qui repose sur une lecture séquentielle du document sous le contrôle de l'application (ce n'est pas le parseur qui pilote le parcours mais l'application qui pilote le parseur). StAX représente un document sous la forme d'un ensemble d'événements qui sont fournis à la demande de l'application dans l'ordre du parcours séquentiel du document.

StAX repose sur le modèle de conception Iterator : chaque élément du document est parcouru séquentiellement à la demande du code pour émettre un événement. Ce parcours se fait à l'aide d'un curseur virtuel.

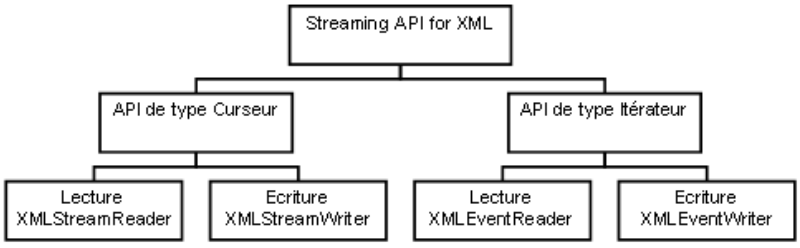
47.2. Les deux API de StAX

StAX est donc une API qui propose de mettre en oeuvre une troisième méthode pour traiter un document XML : le pull parsing. Son but est de fournir un parser qui puisse traiter de gros documents XML, avec une faible quantité de ressources requises et dont les traitements sont contrôlés par le code.

StAX propose une API pour un traitement d'un document XML sous la forme d'une itération sur des événements émis par le parser à la demande du client. Elle propose deux formes d'API :

- une API du type curseur (Cursor) : permet le parcours du document en générant un événement codé sur un entier pour chaque type d'élément rencontré
- une API de type itérateur (Event Iterator) : permet le parcours du document en produisant un événement de type XMLEvent à chaque élément rencontré

Elles permettent toutes les deux la lecture et l'écriture d'un document XML.



La définition de deux API permet de les conserver séparément avec une faible complexité plutôt que d'avoir une seule API plus complexe.

L'API de type curseur parcourt le flux du document et émet des événements sous la forme d'un entier codant chaque événement.

L'API de type curseur est plus efficace dans la mesure où elle n'a pas besoin d'instancier un objet pour chaque événement comme le fait l'API de type itérateur. L'API de type itérateur est plus facile à utiliser puisque toutes les données utiles sont déjà présentes dans l'objet de type XMLEvent.

L'interface XMLStreamReader définit un contrat pour un objet qui va analyser un document XML avec une API de type curseur.

L'interface XMLStreamReader propose des méthodes pour obtenir des informations sur l'élément courant représenté par l'événement courant du curseur. Ces méthodes retournent des chaînes de caractères ce qui limite les ressources à une transformation en chaîne de caractères quels que soient les contenus des éléments retournés.

L'interface XMLStreamWriter définit un contrat pour un objet qui va générer un document XML.

L'API de type itérateur parcourt le flux du document et émet des événements sous la forme d'objets de type XMLEvent qui encapsulent les informations de l'événement.

L'interface XMLEventReader définit un contrat pour un objet qui va analyser un document XML avec une API de type itérateur sur des événements. Elle hérite de l'interface Iterator : elle propose donc la méthode nextEvent() qui retourne le prochain événement et la méthode hasNext() qui permet de savoir s'il y a encore un événement à traiter.

L'interface XMLEvent encapsule les données d'un événement lié au parcours du document XML : ces événements sont émis à la demande du client dans l'ordre de leur apparition lors du parcours du document.

La définition de deux API permet de laisser au développeur le choix d'utiliser l'API de type curseur pour limiter l'instanciation d'objets durant l'analyse du document ou d'utiliser l'API de type itérateur pour bénéficier directement des événements sous la forme d'objets. Le développeur peut

ainsi choisir en fonction de son contexte de mettre en oeuvre l'une ou l'autre des API selon des critères de consommation de ressources ou de simplicité et de fiabilité du code.

L'API de type curseur est moins verbeuse et moins puissante que celle de type itérateur d'événements. Elle est cependant plus efficace car elle instancie moins d'objets. Le code à produire avec l'API de type curseur est plus petit et généralement plus efficace. L'API de type itérateur est plus flexible et évolutive que l'API de type curseur.

Elles permettent toutes les deux uniquement la lecture vers l'avant du document mais l'API de type itérateur propose en plus la méthode peek() qui permet de connaître le prochain événement.

StAX permet aussi de construire un document XML en utilisant les flux. Les API de type curseur et itérateur proposent leur propre interface pour permettre l'écriture de documents.

Les API de StAX sont contenues dans les packages javax.xml.stream et javax.xml.transform.stream

Comme SAX, StAX est un parseur dont les spécifications sont écrites pour Java. Il existe une implémentation de référence mais l'implémentation de ses spécifications peut être réalisée par un tiers.

47.3. Les fabriques

StAX propose des fabriques pour les différents types d'objets : XMLInputFactory, XMLOutputFactory et XMLEventFactory. Des paramètres propres à une implémentation peuvent être manipulés en utilisant les méthodes getProperty() et setProperty() de ces fabriques.

La classe XMLInputFactory est une fabrique qui permet d'obtenir et de configurer une instance du parseur pour une lecture d'un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLInputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLInputFactory du jar
- Utilisation de l'instance par défaut

L'instance de la classe XMLInputFactory permet de configurer et d'instancier un parseur. La configuration se fait en utilisant des propriétés de la fabrique :

Propriété	Rôle
javax.xml.stream.isValidating	Permettre d'activer la validation du document en utilisant sa DTD (optionnelle, false par défaut)
javax.xml.stream.isCoalescing	Permettre d'indiquer si tous les événements de type characters contigus doivent être regroupés en un seul événement (false par défaut)
javax.xml.stream.isNamespaceAware	Supprimer le support des espaces de nommages (optionnelle, true par défaut)
javax.xml.stream.isReplacingEntityReferences	Permettre de demander le remplacement des entités de référence internes par leur valeur et ainsi émettre un événement de type Characters (true par défaut)
javax.xml.stream.isSupportingExternalEntities	
javax.xml.stream.reporter	
javax.xml.stream.resolver	Permettre de préciser une implémentation de type XMLResolver utilisée pour résoudre les entités externes
javax.xml.stream.allocator	
javax.xml.stream.supportDTD	Permettre de préciser si le support des DTD est activé (true par défaut)

La classe XMLOutputFactory est une fabrique qui permet de créer des objets pour écrire un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLOutputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLOutputFactory du jar
- Utilisation de l'instance par défaut

La classe XMLOuputFactory ne propose qu'une seule propriété :

Propriété	Rôle
javax.xml.stream.isRepairingNamespaces	Préciser si un préfixe par défaut doit être créé pour les espaces de nommages

La classe XMLEventFactory est une fabrique qui permet de créer des objets qui héritent de XMLEvent.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLEventFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLEventFactory du jar
- Utilisation de l'instance par défaut

La classe XMLEventFactory ne possède pas de propriétés.

Pour modifier les propriétés de la fabrique, il faut utiliser la méthode setProperty() qui attend en paramètres le nom de la propriété et sa valeur.

Exemple :

```
1. XMLInputFactory xmlif = XMLInputFactory.newInstance();
2.
3. xmlif.setProperty("javax.xml.stream.isCoalescing",Boolean.TRUE);
4. xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences", Boolean.TRUE);
5.
6. XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
7.     "biblio.xml"));
```

Il est important de valoriser les propriétés avant de créer une instance d'un parseur. Une fois cette instance créée, il n'est plus possible de modifier ses propriétés.

Certains paramètres de configuration sont optionnels et ne sont donc pas obligatoirement supportés par une implémentation donnée. La méthode isPropertySupported() des fabriques XMLInputFactory et XMLOuputFactory permet de vérifier le support d'une propriété dont le nom est fourni en paramètre.

Exemple :

```
01. ...
02.       XMLInputFactory xmlif =  XMLInputFactory.newInstance();
03.       if  (xmlif.isPropertySupported("javax.xml.stream.isReplacingEntityReferences"))  {
04.         System.out.println("javax.xml.stream.isReplacingEntityReferences  supporte");
05.         xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences",  Boolean.TRUE);
06.       }
07.       XMLStreamReader xmlsr =  xmlif.createXMLStreamReader(new FileReader(
08.           "biblio.xml"));
09.   ...
```

XMLStreamReader et XMLEventReader possèdent la méthode getProperty() qui permet d'obtenir la valeur d'une propriété.

47.4. Le traitement d'un document XML avec l'API du type curseur

L'API de type curseur ne permet un parcours du document que vers l'avant : l'analyseur StAX parcourt le flux de caractères du document et émet des événements à la demande.

L'interface principale de l'API de type curseur est XMLStreamReader : elle propose des méthodes pour le parcours du document et de nombreuses méthodes qu'il ne faut utiliser que dans le contexte de l'événement en cours de traitement. Les informations retournées par ces méthodes le sont sous la forme de chaînes de caractères directement extraites du document : ceci rend les traitements d'analyse peu consommateurs en ressources.

Lors de l'analyse d'un document, l'instance de l'interface XMLStreamReader permet de se déplacer dans les différents éléments qui composent le document XML en cours de traitement. Ce déplacement ne peut se faire que vers l'avant. Un événement est émis par le parseur à la demande de l'application : celui-ci correspond au type de l'élément courant dans le document.

Il est nécessaire de créer une instance de la classe XMLStreamReader en utilisant la fabrique XMLInputFactory. Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode newInstance().

Exemple :

```
1. XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLStreamReader en utilisant la méthode createXMLStreamReader() de la fabrique. Cette méthode possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
1. XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
2.     "biblio.xml"));
```

Le parseur permet donc d'avancer dans la lecture du document grâce aux méthodes hasNext() et next() de la classe XMLStreamReader qui parcourent séquentiellement les événements émis.

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible pour traitement. La méthode next() permet d'obtenir un identifiant sur l'événement suivant dans le flux de lecture du document.

Ceci permet d'itérer sur les événements jusqu'à ce qu'il n'y en ait plus à traiter. Il suffit pour cela d'appeler la méthode next() tant que hasNext() renvoie true.

Remarque : bien que les méthodes hasNext() et next() soient définies dans l'interface Iterator, l'interface XMLStreamReader n'hérite pas de cette interface.

La mise en oeuvre classique consiste donc à réaliser une itération sur les événements et à exécuter les traitements en fonction de ceux-ci.

Exemple :

```
1. int eventType;
2. while (xmlsr.hasNext()) {
3.   eventType = xmlsr.next();
4.   ...
5. }
```

La méthode next() renvoie un code sous la forme d'un entier qui précise le type d'événement qui a été rencontré lors de la lecture d'un élément du document. Ce code correspond à un type défini sous la forme d'une constante dans l'interface XMLStreamConstants.

Lors du parcours des éléments qui composent le document en cours de traitement, un événement particulier permet de déterminer le type d'élément qui est en cours de traitement. Les événements qui peuvent être retournés par la méthode next() sont :

Evénement	Rôle
START_DOCUMENT	Le début du document (le prologue)
START_ELEMENT	Une balise ouvrante
ATTRIBUTE	Un attribut
NAMESPACE	La déclaration d'un espace de nommage
CHARACTERS	Du texte entre deux balises (pas forcément une balise ouvrante et sa balise fermante)
COMMENT	Un commentaire
SPACE	Un séparateur
PROCESSING_INSTRUCTION	Une instruction de traitement
DTD	Une DTD
ENTITY_REFERENCE	Une entité de référence
CDATA	Une section CData
END_ELEMENT	Une balise fermante
END_DOCUMENT	La fin du document
ENTITY_DECLARATION	La déclaration d'une entité
NOTATION_DECLARATION	La déclaration d'une notation

La méthode getEventType() permet de connaître le type de l'événement courant.

Il est nécessaire de traiter chaque événement en fonction des besoins : généralement un opérateur switch est utilisé pour définir les traitements de chaque événement utile.

Exemple :

```
01. eventType = xmlsr.next();
02. switch (eventType) {
03. case XMLEvent.START_ELEMENT:
04.   System.out.println(xmlsr.getName());
05.   break;
06. case XMLEvent.CHARACTERS:
07.   String chaine = xmlsr.getText();
08.   if (!xmlsr.isWhiteSpace()) {
09.     System.out.println("\t->\"" + chaine + "\"");
10.   }
11.   break;
12. default:
13.   break;
14. }
```

Le premier événement émis lors de l'analyse du document est de type START_DOCUMENT.

A chaque itération, les traitements peuvent prendre en compte ou ignorer l'événement en fonction des besoins. Ceci permet d'avoir une grande liberté sur l'analyse du document.

Exemple :

```
01. package com.jmdoudoux.test.stax;
02.
03. import java.io.FileReader;
04.
05. import javax.xml.stream.XMLInputFactory;
06. import javax.xml.stream.XMLStreamReader;
07. import javax.xml.stream.events.XMLEvent;
08.
09. public class TestStax1 {
10.   public static void main(String args[]) throws Exception {
11.     XMLInputFactory xmlif = XMLInputFactory.newInstance();
12.     XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
13.         "biblio.xml"));
14.     int eventType;
15.     while (xmlsr.hasNext()) {
16.       eventType = xmlsr.next();
17.       switch (eventType) {
18.       case XMLEvent.START_ELEMENT:
19.         System.out.println(xmlsr.getName());
20.         break;
21.       case XMLEvent.CHARACTERS:
22.         String chaine = xmlsr.getText();
23.         if (!xmlsr.isWhiteSpace()) {
24.           System.out.println("\t->\""+chaine+"\"" );
25.         }
26.         break;
27.       default:
28.         break;
29.       }
30.     }
31.   }
32. }
```

Résultat :

```
01. {http://www.jmdoudoux.com/test/jaxb}bibliotheque
02. {http://www.jmdoudoux.com/test/jaxb}livre
03. {http://www.jmdoudoux.com/test/jaxb}titre
04.     ->"titre 1"
05. {http://www.jmdoudoux.com/test/jaxb}auteur
06. {http://www.jmdoudoux.com/test/jaxb}nom
07.     ->"nom 1"
08. {http://www.jmdoudoux.com/test/jaxb}prenom
09.     ->"prenom 1"
10. {http://www.jmdoudoux.com/test/jaxb}editeur
11.     ->"editeur 1"
12. {http://www.jmdoudoux.com/test/jaxb}livre
13. {http://www.jmdoudoux.com/test/jaxb}titre
14.     ->"titre 2"
15. {http://www.jmdoudoux.com/test/jaxb}auteur
16. {http://www.jmdoudoux.com/test/jaxb}nom
17.     ->"nom 2"
18. {http://www.jmdoudoux.com/test/jaxb}prenom
19.     ->"prenom 2"
20. {http://www.jmdoudoux.com/test/jaxb}editeur
21.     ->"editeur 2"
22. {http://www.jmdoudoux.com/test/jaxb}livre
23. {http://www.jmdoudoux.com/test/jaxb}titre
24.     ->"titre 3"
25. {http://www.jmdoudoux.com/test/jaxb}auteur
26. {http://www.jmdoudoux.com/test/jaxb}nom
27.     ->"nom 3"
28. {http://www.jmdoudoux.com/test/jaxb}prenom
29.     ->"prenom 3"
30. {http://www.jmdoudoux.com/test/jaxb}editeur
31.     ->"editeur 3"
```

L'interface XMLStreamReader propose des méthodes pour obtenir des données sur l'élément courant en fonction de l'événement lié à cet élément.

Plusieurs méthodes permettent d'obtenir des informations sur l'élément courant du curseur :
String getName();
String getLocalName();
String getNamespaceURI();
String getText();
String getElementText();
int getEventType();
Location getLocation();
int getAttributeCount();
QName getAttributeName(int);
String getAttributeValue(String, String);

Elle propose plusieurs méthodes pour obtenir des informations sur les attributs :
int getAttributeCount();
String getAttributeNamespace(int index);
String getAttributeLocalName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,String localName);
boolean isAttributeSpecified(int index);

Elle propose plusieurs méthodes pour obtenir des informations sur les espaces de nommages :
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);

Certaines méthodes sont utilisables selon l'événement pour obtenir un complément d'information sur l'entité courante. Ces méthodes ne sont utilisables que dans un contexte précis. Par exemple, la méthode getAttributeValue() n'est utilisable que sur un événement de type START_ELEMENT.

Le tableau ci-dessous précise les méthodes utilisables pour chaque événement :

Evénement	Méthodes
Tous	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(),getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag(), getAttributeXXX(), isAttributeSpecified()
NAMESPACE	next(), nextTag(), getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS	next(), getTextXXX(), nextTag()
CDATA	next(), getTextXXX(), nextTag()
COMMENT	next(), getTextXXX(), nextTag()
SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()
DTD	next(), getText(), nextTag()

Il est préférable d'utiliser la méthode close() de la classe XMLStreamReader à la fin des traitements pour libérer les ressources.

Exemple :

```
1. xmlsr.close();
```

Ci-dessous un exemple complet :

Exemple : le document à traiter

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax" *
03.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.    xsi:schemaLocation="http://www.jmdoudoux.com/test/stax biblio2.xsd  ">
05.   <?MonTraitement?>
06.   <tns:livre>
07.       <!-- mon commentaire -->
08.     <tns:titre>titre 1</tns:titre>
09.     <tns:auteur>
10.       <tns:nom>nom 1</tns:nom>
11.       <tns:prenom>prenom  1</tns:prenom>
12.     </tns:auteur>
13.     <tns:editeur>editeur  1</tns:editeur>
14.   </tns:livre>
15. </tns:bibliotheque>
```

Exemple : parcours du document

```
01. package com.jmdoudoux.test.stax;
02.
```

```java
import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax6 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio2.xml"));
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.START_DOCUMENT:
        System.out.println("START_DOCUMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_DOCUMENT:
        System.out.println("END_DOCUMENT : ");
        break;
      case XMLEvent.COMMENT:
        System.out.println("COMMENT : "+ xmlsr.getText());
        break;
      case XMLEvent.CHARACTERS:
        System.out.println("CHARACTERS : ");
        break;
      case XMLEvent.PROCESSING_INSTRUCTION:
        System.out.println("PROCESSING_INSTRUCTION : "+ xmlsr.getPITarget());
        break;
      default:
        break;
      }
    }
  }
}
```

Résultat d'exécution :

```
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
CHARACTERS :
PROCESSING_INSTRUCTION : MonTraitement
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
COMMENT :  mon commentaire
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
END_DOCUMENT :
```

Chaque événement de type StartElement possède un événement correspondant de type EndElement même si le tag est sous sa forme réduite (<exemple/>)

Par défaut, les attributs n'émettent pas d'événement mais sont accessibles par une collection à partir de l'événement StartElement. Il en est de même avec les espaces de nommages.

Remarque : une partie texte du document peut émettre plusieurs événements de type Characters.

Durant le traitement du document, l'analyseur maintient une pile des espaces de nommages qui sont utilisés. Il est tout à fait possible d'interrompre le parcours du document : c'est un grand avantage de StAX de fournir le contrôle de la progression de l'analyse à l'application.

Par exemple, s'il n'est nécessaire de traiter qu'un seul tag, il suffit de tester la valeur du tag sur un événement de type START_ELEMENT, de réaliser les traitements sur le tag puis d'arrêter le parcours du document.

Exemple :

```
01. package com.jmdoudoux.test.stax;
02.
03. import java.io.FileReader;
04.
05. import javax.xml.stream.XMLInputFactory;
06. import javax.xml.stream.XMLStreamReader;
07. import javax.xml.stream.events.XMLEvent;
08.
09. public class TestStax7 {
10.   public static void main(String args[]) throws Exception {
11.     XMLInputFactory xmlif = XMLInputFactory.newInstance();
12.     XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
13.         "biblio.xml"));
14.     int eventType;
15.     boolean encore = xmlsr.hasNext();
16.
17.     while (encore) {
18.
19.       eventType = xmlsr.next();
20.       if (eventType == XMLEvent.START_ELEMENT) {
21.         System.out.println("element=" + xmlsr.getLocalName());
22.         if (xmlsr.getLocalName().equals("editeur")) {
23.           xmlsr.next();
24.           System.out.println("Premier editeur : " + xmlsr.getText());
25.           encore = false;
26.         }
27.       }
28.       if (!xmlsr.hasNext()) {
29.         encore = false;
30.       }
31.     }
32.   }
33. }
```

Résultat :

```
1. element=bibliotheque
2. element=livre
3. element=titre
4. element=auteur
5. element=nom
6. element=prenom
7. element=editeur
8. Premier editeur : editeur 1
```

47.5. Le traitement d'un document XML avec l'API du type itérateur

L'API de type itérateur repose sur l'interface XMLEventReader qui représente le parseur et sur l'interface XMLEvent qui représente un événement. Ces événements sont réutilisables et peuvent être enrichis avec des événements personnalisés.

L'interface XMLEventReader propose plusieurs méthodes pour itérer sur le document XML et obtenir l'événement courant.

Les événements émis lors de l'analyse du document sont encapsulés dans un objet de type XMLEvent qui possède pour chaque événement une classe fille : Attribute, Characters, Comment, StartDocument, EndDocument, StartElement, EndElement, Namespace, DTD, EntityDeclaration, EntityReference, NotationDeclaration, et ProcessingInstruction. Chacune de ces classes possède des propriétés dédiées.

L'API de type itérateur propose plusieurs types d'événements qui implémentent l'interface XMLEvent :

Evénement	Rôle
StartDocument	Concerne le début du document (le prologue)
StartElement	Concerne le début d'un élément
EndElement	Concerne la fin d'un élément
Characters	Concerne une section de type CData ou une entité de type CharacterEntities. Les séparateurs sont également représentés par cet événement
EntityReference	Concerne une entité de référence
ProcessingInstruction	Concerne une instruction de traitement
Comment	Concerne un tag de commentaires
EndDocument	Concerne la fin du document

DTD	Concerne les informations sur la DTD
Attribut	Normalement les attributs sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement
Namespace	Normalement les espaces de nommages sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement

Remarque : les événements DTD, EntityDeclaration, EntityReférence, NotationDeclaration et ProcessingInstruction ne sont levés que si une DTD est associée au document en cours de traitement.

L'interface StartElement qui hérite de l'interface XMLEvent propose plusieurs méthodes pour obtenir des informations sur les attributs et les espaces de nommages :

- Attribute getAttributeByName() : renvoie un attribut à partir de son nom
- Iterator getAttributes() : permet de parcourir tous les attributs de l'élément
- NamespaceContext getNamespaceContext : renvoie le contexte de l'espace de nommage
- Iterator getNamespaces : permet de parcourir tous les espaces de nommages de l'élément
- String getNamespaceURI : retourne la valeur d'un préfixe dans le contexte de l'élément

L'interface XMLEventReader analyse le document XML et émet des événements sous la forme d'objets de type XMLEvent.

L'utilisation de XMLEventReader est similaire à celle de XMLStreamReader. XMLEventReader permet en plus de connaître le prochain événement grâce à la méthode peek() sans consommer l'événement ce qui permet d'anticiper sur les traitements.

L'interface XMLEventReader définit plusieurs méthodes :

Méthode	Rôle
XMLEvent nextEvent()	obtenir et consommer l'événement suivant (avance dans l'itération)
boolean hasNext()	préciser s'il y a encore un événement à traiter
XMLEvent peek()	obtenir le prochain événement sans le consommer (l'itération ne bouge pas)
next()	avancer dans l'itération et renvoie le prochain événement
XMLEvent nextTag()	obtenir le prochain événement dans l'itération en ignorant les séparateurs pour renvoyer le prochain événement de type START_ELEMENT ou END_ELEMENT

Pour utiliser l'API de type itérateur, plusieurs étapes sont nécessaires.

Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode statique newInstance().

Exemple :

```
1. XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLEventReader en utilisant la méthode createXMLEventReader() de la fabrique qui possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
1. XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
2.     "biblio.xml"));
```

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible. La méthode nextEvent() permet d'obtenir l'événement suivant. Il suffit de faire une itération tant que la méthode hasNext() renvoie true et d'appeler dans cette itération la méthode nextEvent() pour parcourir tout le document.

Exemple :

```
1. XMLEvent event;
2. while (xmler.hasNext()) {
3.   event = xmler.nextEvent();
4.   ...
5. }
```

L'interface XMLEvent propose la méthode getEventType() pour connaître le type de l'événement. Elle propose aussi plusieurs méthodes :

- isXXX() pour chaque événement qui renvoie un booléen indiquant si l'événement est du type XXX.
- asXXX() pour chaque événement qui renvoie une instance de XXX correspondant à l'événement qui est du type XXX.

Exemple complet :

```
01. package com.jmdoudoux.test.stax;
02.
03. import java.io.*;
04. import javax.xml.stream.*;
05. import javax.xml.stream.events.*;
06.
07. public class TestStax2 {
08.
09.   public static void main(String args[]) throws Exception {
10.
11.     XMLInputFactory xmlif = XMLInputFactory.newInstance();
12.     XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
13.         "biblio.xml"));
14.     XMLEvent event;
15.     while (xmler.hasNext()) {
16.       event = xmler.nextEvent();
17.       if (event.isStartElement()) {
18.         System.out.println(event.asStartElement().getName());
19.       } else if (event.isCharacters()) {
20.         if (!event.asCharacters().isWhiteSpace()) {
21.           System.out.println("\t>" + event.asCharacters().getData());
22.         }
23.       }
24.     }
25.   }
26. }
```

Résultat de l'éxécution :

```
01. {http://www.jmdoudoux.com/test/jaxb}bibliotheque
02. {http://www.jmdoudoux.com/test/jaxb}livre
03. {http://www.jmdoudoux.com/test/jaxb}titre
04.     >titre 1
05. {http://www.jmdoudoux.com/test/jaxb}auteur
06. {http://www.jmdoudoux.com/test/jaxb}nom
07.     >nom 1
08. {http://www.jmdoudoux.com/test/jaxb}prenom
09.     >prenom 1
10. {http://www.jmdoudoux.com/test/jaxb}editeur
11.     >editeur 1
12. {http://www.jmdoudoux.com/test/jaxb}livre
13. {http://www.jmdoudoux.com/test/jaxb}titre
14.     >titre 2
15. {http://www.jmdoudoux.com/test/jaxb}auteur
16. {http://www.jmdoudoux.com/test/jaxb}nom
17.     >nom 2
18. {http://www.jmdoudoux.com/test/jaxb}prenom
19.     >prenom 2
20. {http://www.jmdoudoux.com/test/jaxb}editeur
21.     >editeur 2
22. {http://www.jmdoudoux.com/test/jaxb}livre
23. {http://www.jmdoudoux.com/test/jaxb}titre
24.     >titre 3
25. {http://www.jmdoudoux.com/test/jaxb}auteur
26. {http://www.jmdoudoux.com/test/jaxb}nom
27.     >nom 3
28. {http://www.jmdoudoux.com/test/jaxb}prenom
29.     >prenom 3
30. {http://www.jmdoudoux.com/test/jaxb}editeur
31.     >editeur 3
```

Comme avec l'API de type curseur, il est possible avec l'API de type itérateur d'interrompre le traitement du document.

Exemple :

```
01. package com.jmdoudoux.test.stax;
02.
03. import java.io.*;
04. import javax.xml.stream.*;
05. import javax.xml.stream.events.*;
06.
07. public class TestStax3 {
08.
09.   public static void main(String args[]) throws Exception {
10.     boolean termine = false;
11.     XMLInputFactory xmlif = XMLInputFactory.newInstance();
12.     FileReader fr = new FileReader("biblio.xml");
13.     XMLEventReader xmler = xmlif.createXMLEventReader(fr);
14.     XMLEvent event;
15.     termine = !xmler.hasNext();
16.
17.     while (!termine) {
18.       event = xmler.nextEvent();
19.       if (event.isStartElement()) {
20.         if (event.asStartElement().getName().getLocalPart() == "editeur") {
21.           event = xmler.nextEvent();
```

```
            System.out.println("Premier editeur = "+event.asCharacters().getData());
            termine = true;
          }
        }
        if (!termine && !xmler.hasNext()) {
          termine = true;
        }
      }
        fr.close();
        xmler.close();
    }
}
```

Les événements sont émis dans l'ordre de rencontre des éléments lors du parcours du document par le parseur.

Si le document XML est syntaxiquement correct, alors chaque événement de type StartElement possède un événement de type EndElement correspondant.

47.6. La mise en oeuvre des filtres

StAX propose la mise en oeuvre de filtres pour n'obtenir que les événements désirés.

Pour l'API de type itérateur, l'interface EventFilter définit la méthode accept() qui attend en paramètre un objet de type XMLEvent et renvoie un booléen qui précise si cet événement doit être traité.

Exemple :

```
package com.jmdoudoux.test.stax;

import javax.xml.stream.EventFilter;
import javax.xml.stream.events.XMLEvent;

public class MonEventFilter implements EventFilter {

  public boolean accept(XMLEvent event) {

    if (event.isStartElement() || event.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLStreamReader en utilisant la méthode createFilteredReader() de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLStreamReader pour le traitement du document et le filtre.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax11 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();

    XMLStreamReader xmlr = xmlif.createXMLStreamReader(
        new FileReader("biblio.xml"));

    XMLStreamReader xmlsr =
      xmlif.createFilteredReader(xmlr, new MonStreamFilter());

    while (xmlsr.hasNext()) {
      int eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT "+xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT "+xmlsr.getName());
        break;
      default:
        System.out.println("AUTRE "+xmlsr.getName());
        break;
      }
    }
```

```
  }
}
```

Résultat :

```
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

Pour l'API de type curseur, l'interface StreamFilter définit la méthode accept() qui attend en paramètre un objet de type XMLStreamReader et renvoie un booléen qui précise si l'événement courant doit être traité.

Exemple :

```java
package com.jmdoudoux.test.stax;

import javax.xml.stream.StreamFilter;
import javax.xml.stream.XMLStreamReader;

public class MonStreamfilter implements StreamFilter {

  public boolean accept(XMLStreamReader reader) {
    if(reader.isStartElement() || reader.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLEventReader en utilisant la méthode createFilteredReader de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLEventReader pour le traitement du document et le filtre.

Exemple :

```java
package com.jmdoudoux.test.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStAX12 {

  public static void main(String args[]) throws Exception {

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLEventReader xmlr = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
    XMLEventReader xmler = xmlif.createFilteredReader(xmlr,
        new MonEventFilter());

```

```
    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        System.out.println("StartElement=" + event.asStartElement().getName());
      } else if (event.isEndElement()) {
        System.out.println("EndElement=" + event.asEndElement().getName());
      } else {
        System.out.println("Autre");
      }
    }
  }
}
```

Résultat :

```
StartElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
EndElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

47.7. L'écriture un document XML avec l'API de type curseur

L'interface XMLStreamWriter propose des fonctionnalités simples et de bas niveau pour écrire un document.

L'interface XMLStreamWriter définit les méthodes pour un objet capable de réécrire un document en cours de parcours ou d'écrire un nouveau document.

Une instance d'un tel objet est obtenue en utilisant la fabrique XMLOutputFactory.

Exemple :

```
XMLStreamWriter writer = XMLOutputFactory.newInstance().
    createXMLStreamWriter(outStream);
```

L'interface XMLStreamWriter propose de nombreuses méthodes pour ajouter des noeuds de différents types au document en cours de rédaction :

Méthode	Rôle
writeStartDocument()	ajouter le prologue du document
writeEndDocument()	ajouter tous les éléments de type fin requis pour terminer le document
writeStartElement()	ajouter un élément de type début
writeEndElement()	ajouter un élément de type fin

writeComment()	ajouter un élément de type commentaire
writeNamespace()	ajouter un espace de nommage
writeCharacters()	ajouter un élément de type texte
writeProcessingInstruction()	ajouter une instruction de traitement

Remarque : chaque méthode writeStartXxx() doit avoir un appel à la méthode writeEndXxx() correspondante dans les traitements.

Il faut obtenir une instance de la fabrique XMLOuputFactory en utilisant sa méthode newInstance().

Exemple :

```
1. XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
```

Il faut instancier un objet de type FileWriter qui va encapsuler le fichier où sera stocké le document XML

Exemple :

```
1. FileWriter output = new FileWriter(new File("test.xml"));
```

Il faut obtenir une instance de l'interface XMLStreamWriter en utilisant la méthode createXMLStreamWriter() de la fabrique.

Exemple :

```
1. XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
```

Il faut créer le prologue du document en utilisant la méthode writeStartDocument() qui attend en paramètre le nom du jeu de caractères d'encodage et la version de xml. Ces deux informations ne sont utilisées que comme valeurs des attributs encoding et version du prologue.

Exemple :

```
1. xmlsw.writeStartDocument("Cp1252", "1.0");
```

Pour préciser le jeu de caractères utilisé pour encoder le document XML, il est nécessaire d'utiliser une version surchargée de la méthode createXMLStreamWriter().

Exemple :

```
1. FileOutputStream output = new FileOutputStream("test.xml");
2. XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output, "UTF-8");
3. xmlsw.writeStartDocument("UTF-8", "1.0");
```

La création d'une balise dans le document à la position courante se fait en utilisant la méthode writeStartElement(). Cette méthode possède trois surcharges qui permettent de préciser le nom de la balise, son préfixe et l'URI de son espace de nommage.

La méthode writeNamespace() qui attend en paramètres un préfixe et une uri permet de définir un espace de nommage pour la balise courante.

Exemple :

```
1. xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
```

La méthode writeAttribut() permet de définir un attribut pour la balise courante. Elle possède plusieurs surcharges qui attendent en paramètres le nom de l'attribut, sa valeur, un préfixe et l'uri de l'espace de nommage

Exemple :

```
1. xmlsw.writeAttribut("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

La méthode writeCharacters() qui peut être utilisée avec une chaîne de caractères ou un tableau de caractères permet d'écrire un noeud de type texte dans la balise courante.

Exemple :

```
1. xmlsw.writeCharacters("titre "+i);
```

La méthode writeCharacters() permet d'ajouter du texte dans le document en échappant les caractères utilisés par XML (<, >, &, ...).

La méthode writeEndElement() permet de créer un balise fermante à la balise courante. Elle détermine automatiquement le nom de la balise courante pour créer la balise nécessaire. Son appel est obligatoire pour chaque balise ouverte.

Exemple :

```
1. xmlsw.writeEndElement();
```

Une balise de commentaires peut être créée en utilisant la méthode writeComment().

Exemple :

```
1. xmlsw.writeComment("Fichier de test XMLStreamWriter");
```

La méthode writeProcessingInstruction() permet d'ajouter une balise de type instruction de traitement.

La méthode writeEndDocument() permet de créer toutes les balises fermantes requises à partir de la balise courante jusqu'à la balise racine.

Une fois le document complet, il est nécessaire d'utiliser les méthodes flush() et close() de la classe XMLStreamWriter pour enregistrer le document XML dans le fichier.

Exemple :

```
1. xmlsw.flush();
2. xmlsw.close();
```

Exemple complet :

```
01. package com.jmdoudoux.test.stax;
02.
03. import java.io.StringWriter;
04.
05. import javax.xml.stream.XMLOutputFactory;
06. import javax.xml.stream.XMLStreamWriter;
07.
08. public class TestStax5 {
09.
10.   public static void main(String args[]) throws Exception {
11.
12.     String ns = "http://www.jmdoudoux.com/test/stax";
13.
14.     StringWriter strw = new StringWriter();
15.     XMLOutputFactory output = XMLOutputFactory.newInstance();
16.     XMLStreamWriter writer = output.createXMLStreamWriter(strw);
17.     writer.writeStartDocument();
18.     writer.setPrefix("tns",ns);
19.     writer.setDefaultNamespace(ns);
20.     writer.writeStartElement(ns,"bibliotheque");
21.       writer.writeNamespace("tns",ns);
22.       writer.writeStartElement(ns,"livre");
23.         writer.writeAttribute("id","1");
24.         writer.writeStartElement(ns,"titre");
25.           writer.writeCharacters("titre1");
26.         writer.writeEndElement();
27.         writer.writeStartElement(ns,"auteur");
28.           writer.writeStartElement(ns, "nom");
29.           writer.writeCharacters("nom1");
30.           writer.writeEndElement();
31.           writer.writeStartElement(ns,"prenom");
32.           writer.writeCharacters("prenom1");
33.           writer.writeEndElement();
34.         writer.writeEndElement();
35.         writer.writeStartElement(ns,"editeur");
36.         writer.writeCharacters("editeur1");
37.         writer.writeEndElement();
38.       writer.writeEndElement();
39.     writer.writeEndElement();
40.     writer.flush();
41.
42.     System.out.println(strw.toString());
43.   }
44. }
```

Remarque : l'indentation des méthodes writeXxx() permet de vérifier qu'aucun appel de méthode n'a été oublié.

Résultat :

```
<?xml version="1.0" ?>
<bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
   <tns:livre id="1">
     <tns:titre>titre1</tns:titre>
     <tns:auteur>
       <tns:nom>nom1</nom>
       <tns:prenom>prenom1</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur1</tns:editeur>
  </tns:livre>
</bibliotheque>
```

Attention : une implémentation de l'interface XMLStreamWriter n'a pas l'obligation de vérifier que le document créé soit bien formé. Par exemple, l'oubli d'un appel à la méthode writeEndElement() pour un tag provoque un décalage dans la balise de fin, il en résulte l'absence de la balise de fermeture du tag racine.

Exemple complet :

```
package com.jmdoudoux.test.stax;

import java.io.File;
import java.io.FileWriter;

import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class TestStax4 {

  public static void main(String args[]) throws Exception {

    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    FileWriter output = new FileWriter(new File("test.xml"));
    XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
    xmlsw.writeStartDocument("Cp1252", "1.0");
    xmlsw.writeComment("Fichier de test XMLStreamWriter");
    xmlsw.writeStartElement("tns", "bibliotheque",
        "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("xsi", "http://www.w3.org/2001/XMLSchema-instance");
    xmlsw.writeAttribute("xsi:schemaLocation",
        "http://www.jmdoudoux.com/test/stax/biblio.xsd");

    for (int i = 1; i > 4; i++) {

      xmlsw.writeStartElement("tns", "livre",
          "http://www.jmdoudoux.com/test/stax");

      xmlsw.writeStartElement("tns", "titre",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("titre "+i);
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "auteur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeStartElement("tns", "nom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("nom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeStartElement("tns", "prenom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("prenom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "editeur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("editeur "+i);
      xmlsw.writeEndElement();

      xmlsw.writeEndElement();
    }

    xmlsw.writeEndElement();
    xmlsw.flush();
    xmlsw.close();

  }
}
```

Résultat :

```
<?xml  version="1.0" encoding="Cp1252"?><!--Fichier de test  XMLStreamWriter-->
```

```xml
<tns:bibliotheque xmlns:tns="http://www.jmdoudoux.com/test/stax"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jmdoudoux.com/test/stax/biblio.xsd">
 <tns:livre>
    <tns:titre>titre 1</tns:titre>
    <tns:auteur>
      <tns:nom>nom 1</tns:nom>
      <tns:prenom>prenom 1</tns:prenom>
    </tns:auteur>
    <tns:editeur>editeur 1</tns:editeur>
  </tns:livre>
  <tns:livre>
    <tns:titre>titre 2</tns:titre>
    <tns:auteur>
      <tns:nom>nom 2</tns:nom>
      <tns:prenom>prenom 2</tns:prenom>
    </tns:auteur>
    <tns:editeur>editeur 2</tns:editeur>
  </tns:livre>
  <tns:livre>
    <tns:titre>titre 3</tns:titre>
    <tns:auteur>
      <tns:nom>nom 3</tns:nom>
      <tns:prenom>prenom 3</tns:prenom>
    </tns:auteur>
    <tns:editeur>editeur 3</tns:editeur>
  </tns:livre>
</tns:bibliotheque>
```

47.8. L'écriture un document XML avec l'API de type itérateur

L'interface XMLEventWriter propose des fonctionnalités à l'API de type itérateur pour écrire un document XML : celle-ci est particulièrement adaptée à la réécriture d'un document en cours de traitement par l'API de type itérateur mais elle peut aussi être utilisée pour créer un nouveau document. Elle propose des méthodes pour créer un document XML à partir d'objets de type XMLEvent.

Une instance de type XMLEventWriter est obtenue en utilisant la fabrique XMLOutPutFactory.

Elle possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
void flush()	Permettre de vider le cache et d'écrire les données qu'il contient
void close()	Fermer le flux d'écriture
void add(XMLEvent)	Ajouter un élément dans le document

Les événements sont ajoutés au fur et à mesure et ne peuvent plus être modifiés une fois ajoutés. L'ajout d'attributs ou d'espaces de nommages se fait toujours sur le dernier élément de type StartElement ajouté dans le flux.

La méthode setPrefix() permet d'associer un préfixe à un espace de nommage.

Il faut instancier une occurrence de l'interface XMLEventWriter à partir d'une fabrique de type XMLOutputFactory.

Exemple :

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
    "test2.xml"));
```

Il faut obtenir une instance de la fabrique XMLEventFactory en utilisant sa méthode newInstance().

Exemple :

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

Cette fabrique permet de créer des instances des événements qui seront ajoutés dans le document.

Exemple :

```
writer.add(eventFactory.createStartDocument());
```

Une fois le document terminé, il suffit d'appeler les méthodes flush() et close().

Exemple :

```java
package com.jmdoudoux.test.stax;

import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLOutputFactory;

public class TestStax8 {

  private static final String NS_TNS = "http://www.jmdoudoux.com/test/stax";

  private static final String PREFIX_TNS = "tns";

  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test2.xml"));
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();

    writer.setPrefix(PREFIX_TNS, NS_TNS);
    writer.add(eventFactory.createStartDocument());
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS,
        "bibliotheque"));
    writer.add(eventFactory.createNamespace(PREFIX_TNS, NS_TNS));
    writer.add(eventFactory.createProcessingInstruction("MonTraitement", ""));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory.createComment("mon commentaire"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createCharacters("titre 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createCharacters("nom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createCharacters("prenom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createCharacters("editeur 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory
        .createEndElement(PREFIX_TNS, NS_TNS, "bibliotheque"));

    writer.add(eventFactory.createEndDocument());
    writer.flush();
    writer.close();

  }
}
```

Résultat :

```
<?xml version="1.0"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
  <?MonTraitement  ?>
    <tns:livre>
      <!--mon commentaire-->
      <tns:titre>titre  1</tns:titre>
      <tns:auteur>
        <tns:nom>nom  1</tns:nom>
        <tns:prenom>prenom  1</tns:prenom>
      </tns:auteur>
    <tns:editeur>editeur  1</tns:editeur>
  </tns:livre>>/p<
</tns:bibliotheque>
```

Il est aussi possible d'utiliser l'API de type itérateur en lecture et en écriture simultanément.

Exemple :

```java
package com.jmdoudoux.test.stax;

import java.io.FileReader;
import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.events.Characters;
import javax.xml.stream.events.XMLEvent;

```

```java
public class TestStax9 {

  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    FileReader fr = new FileReader("biblio.xml");
    XMLEventReader reader = xmlif.createXMLEventReader(fr);

    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test3.xml"));

    while (reader.hasNext()) {
      XMLEvent event = (XMLEvent) reader.next();
      if (event.getEventType() == XMLEvent.CHARACTERS) {
        Characters characters = event.asCharacters();
        if (!characters.isWhiteSpace()) {
          writer.add(eventFactory.createCharacters(characters.getData() + " modif"));
        }
      } else {
        writer.add(event);
      }
    }
    writer.flush();

    writer.close();

  }
}
```

Résultat :

```xml
<?xml  version="1.0"?>
  <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jmdoudoux.com/test/jaxb _; biblio.xsd ">
     <tns:livre>
       <tns:titre>titre 1 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 1 modif</tns:nom>
         <tns:prenom>prenom 1 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 1 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 2 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 2 modif</tns:nom>
         <tns:prenom>prenom 2 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 2 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 3 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 3 modif</tns:nom>
         <tns:prenom>prenom 3 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 3 modif</tns:editeur>
     </tns:livre>
</tns:bibliotheque>
```

47.9. La comparaison entre SAX, DOM et StAX

Les parseurs avant l'arrivée de StAX utilisent deux méthodes principales pour traiter un document XML :

- ceux basés sur un modèle événementiel utilisé par SAX notamment
- ceux basés sur un modèle reposant sur un arbre d'objets utilisé par DOM notamment

Ces deux modèles ont chacun leurs avantages et leurs inconvénients.

	Avantages	Inconvénients
SAX	• grande efficacité • faible ressource nécessaire • API simple • traitement au fur et à mesure de la lecture	• l'état du document doit être conservé « manuellement » • le document tout entier doit être parcouru • ne peut être utilisé que pour lire un document • traitements séquentiels
DOM	• lecture aléatoire dans l'arbre du document • permet la mise à jour d'un document	• ressources nécessaires importantes proportionnelles à la taille du document • API plus complexe car non développée spécifiquement pour Java

Les trois API de JAXP permettant d'analyser un document XML ont chacune des points forts et des points faibles dont il faut tenir compte pour déterminer quelle API sera la mieux adaptée en fonction des besoins.

	SAX	StAX	DOM
Type de traitements	Événement de type push	Événement de type pull	Arbre d'objets en mémoire
Facilité de mise en oeuvre	Moyenne	Elevée	Moyenne
Support XPath	Non	Non	Oui
Consommation en ressources	Faible	Faible	Dépendante de la taille du document
Sens de parcours	Vers l'avant uniquement	Vers l'avant uniquement	Libre
Lecture	Oui	Oui	Oui
Ecriture	Non	Oui	Oui
Modification	Non	Non	Oui

Même si l'API StAX est basée sur des événements, ses fonctionnalités la placent entre les deux autres types de parsers.

SAX et StAX reposent tous les deux sur un traitement par flux : le document est parcouru et traité au fur et à mesure. Ce type de traitement est efficace et peu consommateur en ressources : il est donc particulièrement adapté au traitement de gros documents.

L'avantage de StAX par rapport à SAX est de donner la possibilité au développeur de demander le prochain événement et de le traiter si nécessaire plutôt que de fournir des traitements dans des fonctions de type « callback » appelées par le parseur. Ceci donne au développeur un meilleur contrôle sur les traitements en facilitant leur mise en oeuvre et permet à tout moment d'interrompre le traitement du parseur sans attendre le traitement de tout le document.

SAX lit et analyse le document au fur et à mesure et émet des événements à destination d'un handler définit dans l'application qui est composée de méthodes de type callback. Ces méthodes sont automatiquement exécutées par le parseur en fonction des événements émis par ce dernier lors de la lecture du document. C'est donc le parseur qui a le contrôle sur les traitements d'analyse du document : ce type de traitement est dit push (c'est le parseur qui émet des événements à son initiative vers l'application). Il nécessite le parcours de tout le document. SAX ne permet pas d'écrire un document XML.

SAX n'est pas aussi simple à mettre en oeuvre que StAX puisqu'il faut développer un handler qui va traiter les événements émis sous la forme de callback : le code des traitements pour sa mise en oeuvre peut être rapidement complexe. Stax est plus simple que SAX : c'est une forme de traitement de type pull (les événements sont émis par le parseur à la demande de l'application) ce qui permet de donner le contrôle de l'analyse au développeur grâce à un parcours d'un ensemble d'événements.

Avec StAX, c'est donc l'application qui possède le contrôle sur le traitement du document ce qui rend plus intuitif le code à écrire pour traiter le document : l'application peut ignorer un élément, appliquer un filtre ou arrêter le traitement du document à tout moment.

Les fonctionnalités de StAX sont proches de celles de SAX. Cependant StAX propose des fonctionnalités supplémentaires :

- une mise en oeuvre des traitements sous une forme itérative qui la rend plus naturelle que la forme de type callback de SAX
- StAX permet l'écriture de documents
- StAX peut être plus efficace car il n'oblige pas à traiter tout le document

StAX est donc aussi efficace que SAX en proposant un modèle de mise en oeuvre plus facile et extensible. StAX pourrait remplacer SAX mais StAX est une API récente qui ne possède pas d'implémentation dans d'autres langages pour le moment. SAX est un standard de fait implémenté dans de nombreuses solutions de parsing dans différentes plates-formes et langages.

DOM est basé sur un arbre d'objets en mémoire qui représente l'ensemble des éléments d'un document XML. Ceci est très pratique pour permettre de se déplacer librement dans le document et de le parcourir à son gré d'autant que DOM supporte l'utilisation des expressions XPath.

DOM est la seule API qui permet de modifier le document. La contre-partie est que DOM consomme beaucoup de ressources et notamment de mémoire puisque tout le document est représenté par un arbre d'objets en mémoire : cela exclut de fait son utilisation pour des documents XML volumineux.

DOM est donc l'API la plus puissante puisqu'elle permet un parcours du document dans n'importe quel ordre et quel sens, de modifier le document (création, modification et suppression de noeuds dans le document) et de l'écrire.

DOM et StAX ont en commun de pouvoir écrire un document XML.

L'existence de trois API pour traiter un document XML entraîne logiquement des interrogations sur le choix de l'API à utiliser en fonction du besoin. Il n'existe pas de règles immuables concernant ces choix mais voici quelques cas d'utilisation particuliers :

- La transformation d'un document XML en un autre document XML : il est fréquent de devoir transformer un document XML en un autre pour modifier sa structure ou l'appauvrir. La solution la plus adaptée pour modifier la structure ou appauvrir le document semble être XSLT puisque c'est son rôle principal. StAX ou DOM peuvent être aussi utilisés notamment dans le cas d'enrichissement du document : ces deux API nécessitent l'écriture de code mais cela permet aussi un accès à toutes les API de Java.
- Data Binding : l'utilisation de plus en plus fréquente de XML nécessite de pouvoir mapper un objet à un document ou une portion de document XML et vice versa. Le plus simple est d'utiliser une API dédiée telle que JAXB mais il est aussi possible de réaliser ce traitement à la main. SAX ne peut être utilisé que pour mapper un document XML dans un objet (unmarshalling). StAX et DOM pouvant écrire un document, ces deux API peuvent être utilisés pour des opérations dans les deux sens.
- Un document XML comme source de données : les données à utiliser et éventuellement à mettre à jour sont stockées dans un document XML. Dans ce cas de figure, seul DOM peut répondre au besoin grâce à son support de XPath pour accéder directement à une donnée et sa possibilité de modifier le contenu du document pour réaliser des mises à jour.

StAX peut donc être utilisé dans de nombreux cas de traitements de documents XML.