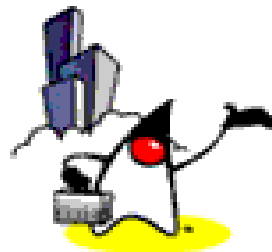




StAX (Streaming API for XML)

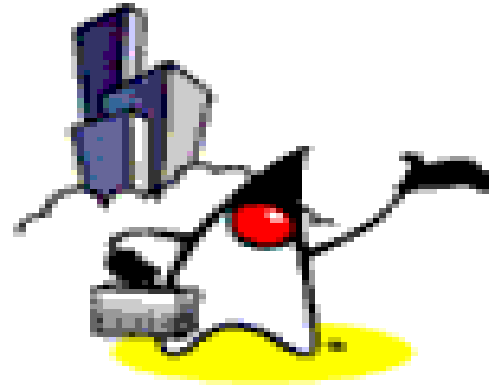


Topics

- Why StAX?
- Iterator API
 - XMLEventReader, XMLEventWriter
- Cursor API
 - XMLStreamReader, XMLStreamWriter
- Filter API
- Choosing between Cursor and Iterator API's

What is StAX?

- It is a streaming (as opposed to in-memory tree) Java-based, event-driven, pull-parsing API (as opposed to push-parsing) for reading and writing XML documents.
- StAX enables you to create bidirectional XML parsers that are fast, relatively easy to program, and have a light memory footprint.
- StAX is the latest API in the JAXP family, and provides an alternative to SAX and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.



Why StAX?

Pull Parsing vs. Push Parsing

- Pull parsing refers to a programming model in which a client application calls methods on an XML parsing library when it needs to interact with an XML infoset--that is, **the client only gets (pulls) XML data when it explicitly asks for it.**
- Push parsing refers to a programming model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML infoset--that is, **the parser sends the data whether or not the client is ready to use it at that time.**

Advantages of Pull Parsing

- With pull parsing, the client has the complete control
 - The client can start, proceed, pause, and resume the parsing process
 - By contrast, with push processing, the parser controls the application thread

Advantages of Pull Parsing

- Pull clients can read multiple documents at one time with a single thread.
- A StAX pull parser can filter XML documents such that elements unnecessary to the client can be ignored
 - In push parsing, application has to receive all elements since there is no filtering scheme
- Pull parsing is easier to use than DOM for writing out

Why StAX?

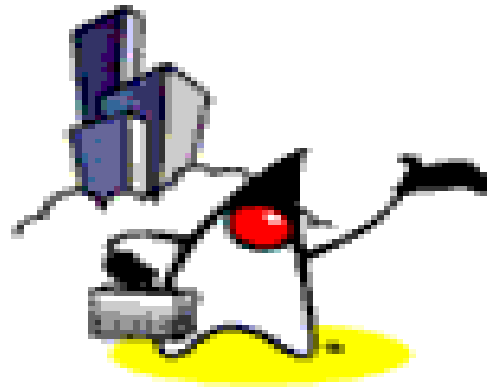
- Supports Pull parsing
- Gives "parsing control" to the programmer
 - This allows the programmer to ask for the next event (pull the event) and allows state to be stored in procedural fashion.
- StAX was created to address limitations in SAX and DOM

StAX vs SAX

- StAX is pull parsing while SAX is push parsing
- StAX-enabled clients are generally easier to code than SAX clients
- StAX is a bidirectional API, meaning that it can both read and write XML documents.
 - SAX is read only, so another API is needed if you want to write XML documents.

Overall Comparison among Parsing APIs

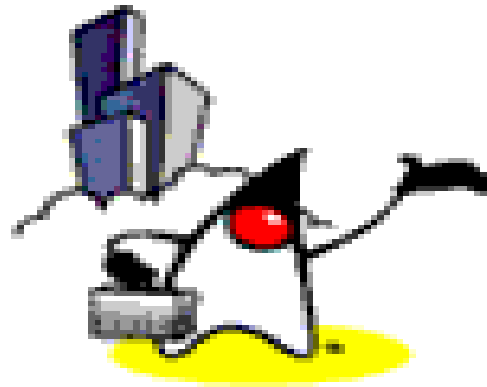
Feature	StAX	SAX	DOM	TrAX
API Type	Pull, streaming	Push, streaming	In memory tree	XSLT Rule
Ease of Use	High	Medium	High	Medium
XPath Capability	No	No	Yes	Yes
CPU and Memory Efficiency	Good	Good	Varies	Varies
Forward Only	Yes	Yes	No	No
Read XML	Yes	Yes	Yes	Yes
Write XML	Yes	No	Yes	Yes
Create, Read, Update, Delete	No	No	Yes	No



Two Types of API

Two Types of StAX API

- Iterator API
 - Convenient, easy to use
- Cursor API
 - Fast, low-level



Iterator API

Iterator API

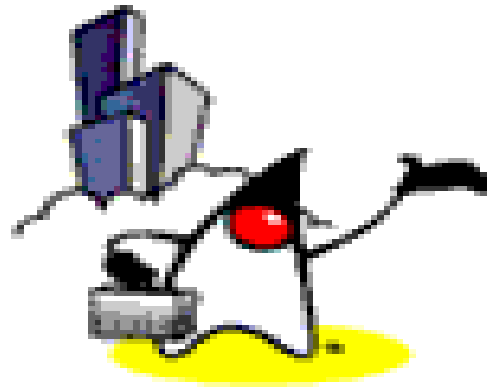
- Represents an XML document stream as a set of discrete event objects.
- These events are **pulled by the application** and provided by the parser in the order in which they are read in the source XML document.

Iterator API Classes

- XMLEvent
- XMLEventReader
- XMLEventWriter

XMLEvent Types

- StartDocument
- StartElement, EndElement, Characters
- EntityReference, ProcessingInstruction
- Comment, EndDocument, DTD
- Attribute, Namespace



XMLEventReader API

XMLStreamReader

```
public interface XMLStreamReader extends Iterator {  
    public XMLEvent nextEvent() throws XMLStreamException;  
    public boolean hasNext();  
    public XMLEvent peek() throws XMLStreamException;  
    ...  
}
```

XMLEventReader (from “event” example app)

```
//Get the factory instace first.
```

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

```
System.out.println("FACTORY: " + factory);
```

```
//create the XMLEventReader, pass the filename for any relative resolution
```

```
XMLEventReader r = factory.createXMLEventReader(
```

```
    filename,
```

```
    new FileInputStream(filename));
```

```
// Programmer asks for events when s/he wants it (as opposed to
```

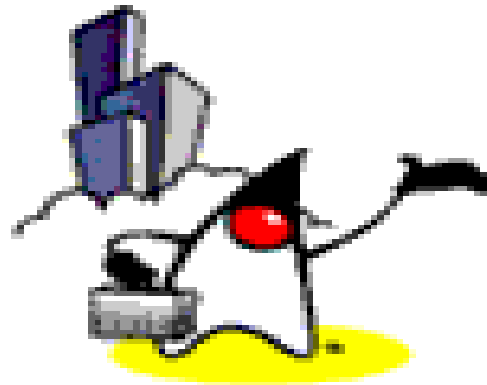
```
// given by parser as in the case of SAX)
```

```
while (r.hasNext()) {
```

```
    XMLEvent e = r.nextEvent();
```

```
    System.out.println("Event -> " + e.toString());
```

```
}
```



XMLEventWriter API

XMLEventWriter

- Stax has writing APIs
 - The XMLEventWriter class extends from XMLEventConsumer interface
- XMLEventWriter acts as a consumer which can consume events
- Event producer XMLEventReader and consumer XMLEventWriter mechanism makes it possible to read XML from one stream sequentially and simultaneously write to other stream

XMLStreamWriter

```
public interface XMLStreamWriter {  
    public void flush() throws XMLStreamException;  
    public void close() throws XMLStreamException;  
    public void add(XMLEvent e) throws XMLStreamException;  
    public void add(Attribute attribute) throws XMLStreamException;  
    ...  
}
```

XMLEventWriter (from “readwrite” example app)

```
EventProducerConsumer ms = new EventProducerConsumer();
XMLEventReader reader = XMLInputFactory.newInstance()
    .createXMLEventReader(new java.io.FileInputStream(args[0]));
XMLEventWriter writer = XMLOutputFactory.newInstance()
    .createXMLEventWriter( System.out);

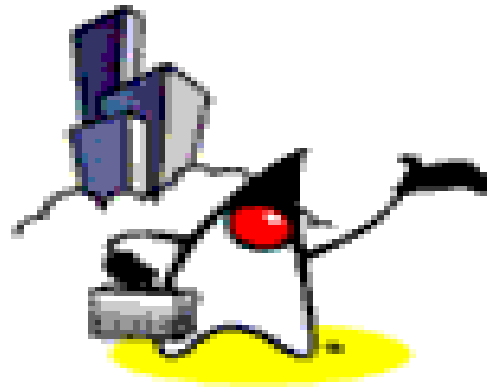
while (reader.hasNext()) {
    XMLEvent event = (XMLEvent) reader.next();

    //write this event to Consumer (XMLOutputStream)
    if (event.getEventType() == event.CHARACTERS) {
        // See the code of getNewCharactersEvent(..) in next slide
        writer.add(ms.getNewCharactersEvent(event.asCharacters()));
    } else {
        writer.add(event);
    }
}
writer.flush();

// See next slide for getNewCharactersEvent(..)
```

XMLEventWriter (from “readwrite” example app)

```
/** New Character event (with text containing current time) is
 * created using XMLEventFactory in case the
 * Characters event passed matches the criteria.
 *
 * @param Characters Current character event.
 * return Characters New Characters event.
 */
Characters getNewCharactersEvent(Characters event) {
    if (event.getData().equalsIgnoreCase("The First and Last Freedom")) {
        return m_eventFactory.createCharacters(
            Calendar.getInstance().getTime().toString());
    } //else return the same event
    else {
        return event;
    }
}
```



Cursor API

Caveat of Iterator API

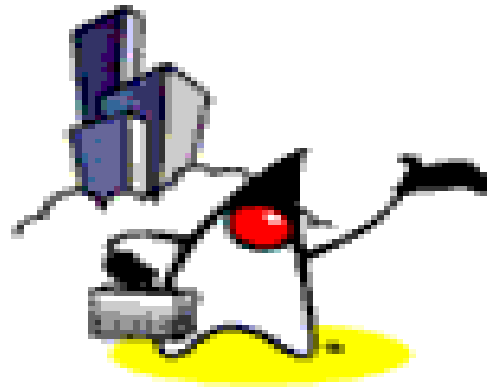
- While the iterator-style API is convenient and easy to use, it involves some overhead
 - The parser needs to create event objects

Cursor API over Iterator API

- For applications where high performance is paramount, you may want to use the cursor-based API instead
- The type *XMLStreamReader* features a *next()* method that delivers integer values (instead of event objects) representing the event type.

Cursor API

- Represents a cursor with which you can walk an XML document from beginning to end.
- This cursor can point to one thing at a time, and always moves forward, never backward, usually one infoset element at a time



XMLStreamReader

Cursor API - XMLStreamReader

- XMLStreamReader
 - Includes accessor methods for all possible information retrievable from the XML Information model including document encoding, element names, attributes, namespaces, text nodes, start tags, comments, processing instructions, document boundaries, and so forth
- XMLStreamWriter
 - Provides methods that correspond to StartElement and EndElement event types

XMLStreamReader Methods

```
public QName getName()
public String getLocalName()
public String getNamespaceURI()
public String getText()
public String getElementText()
public int getEventType()
public Location getLocation()
public int getAttributeCount()
public QName getAttributeName(int index)
public String getAttributeValue(String namespaceURI,
    String localName)
// There are more
```

Caveats

- Not all of the getter methods work all the time
 - For instance, if the cursor is positioned on an end-tag, then you can get the name and namespace but not the attributes or the element text.
 - If the cursor is positioned on a text node, then you can get the text but not the name, namespace, prefix, or attributes. Text nodes just don't have these things.
- Calling an inapplicable method normally returns null.

getEventType() method

- To find out what kind of node the parser is currently positioned on, you call the *getEventType()* method, which returns one of these seventeen int constants:
 - XMLStreamConstants.START_DOCUMENT
 - XMLStreamConstants.END_DOCUMENT
 - XMLStreamConstants.START_ELEMENT
 - XMLStreamConstants.END_ELEMENT
 - XMLStreamConstants.ATTRIBUTE
 - XMLStreamConstants.CHARACTERS
 - XMLStreamConstants.CDATA
 - XMLStreamConstants.SPACE
 - ...

Example: XMLStreamReader (from “cursor” example app)

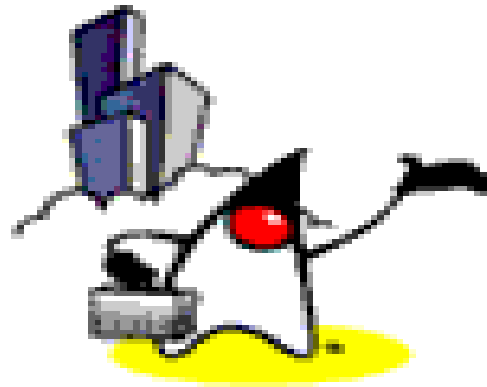
```
XMLStreamReader parser = xmlif.createXMLStreamReader(
    filename,
    new FileInputStream(filename));
while (true) {
    int event = parser.next();
    if (event == XMLStreamConstants.END_DOCUMENT) {
        parser.close();
        break;
    }
    if (event == XMLStreamConstants.START_ELEMENT) {
        System.out.println(parser.getLocalName());
    }
}
```

How to use XMLStreamReader

- Since the client application controls the process, it's easy to write separate methods for different elements
- For example, you could write one method that handles headers, one that handles img elements, one that handles tables, one that handles meta tags, and so forth.

Example: Usage pattern

```
// Process an html element that contains head and
// body child elements
public void processHtml(XmlPullParser parser) {
    while (true) {
        int event = parser.next();
        if (event == XMLStreamConstants.START_ELEMENT) {
            if (parser.getLocalName().equals("head")) processHead(parser);
            else if (parser.getLocalName().equals("body")) processBody(parser)
        }
        else if (event == XMLStreamConstants.END_ELEMENT) {
            return;
        }
    }
}
```



XMLStreamWriter

XMLStreamWriter

```
public interface XMLStreamWriter {  
    public void writeStartElement(String localName)  
        throws XMLStreamException;  
    public void writeEndElement()  
        throws XMLStreamException;  
    public void writeCharacters(String text)  
        throws XMLStreamException;  
    // ... other methods not shown  
}
```

Ease of Development

- Think XML

```
XMLStreamWriter.  
writeStartDocument(  
    "utf-8", "1.0");  
writeStartElement(  
    "hello");  
writeDefaultNamespace("h  
    ttp://samples");  
writeCharacters(  
    "this crazy");  
writeEmptyElement(  
    "world");  
writeEndElement();  
writeEndDocument();
```

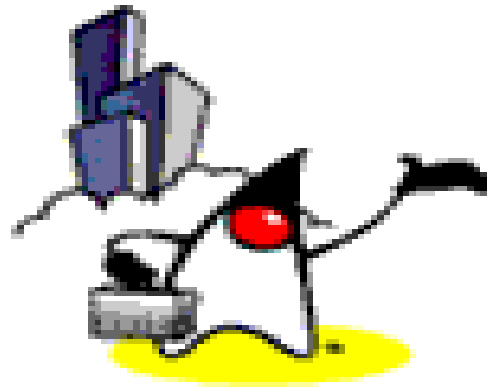
- For XML

```
<?xml  
    version="1.0"  
    encoding="utf-  
    8"?>  
<hello  
    xmlns="http://sa  
    mples">this  
    crazy<world/>  
</hello>
```

Example: XMLStreamWriter (from writer example app)

```
XMLStreamWriter writer = xmlif.createXMLStreamWriter(  
    filename,  
    new FileOutputStream(filename));
```

```
writer.writeStartDocument("ISO-8859-1", "1.0");  
writer.writeStartElement("greeting");  
writer.writeAttribute("id", "g1");  
writer.writeCharacters("Hello StAX");  
writer.writeEndDocument();
```



StreamFilter API

StreamFilter

- Stream through the XML and only pay attention to the ones I care
 - Elements
 - Namespace
- Ease of development
- Performance
 - Lower level filtering
 - Stream dances lightly, quickly, efficiently

StreamFilter Class

```
accept(XMLStreamReader reader) {  
    // Filtering code  
}
```

StreamFilter Example #1

```
//Accept only StartElement and EndElement events,  
//Filters out rest of the events.
```

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

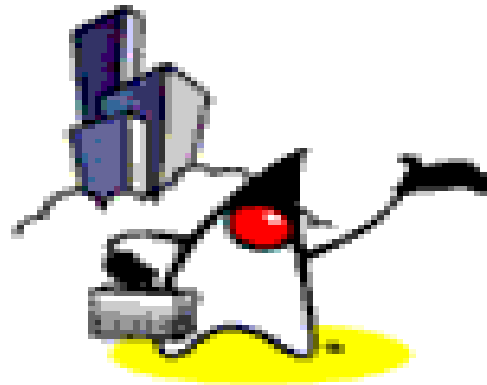
StreamFilter Example #2

```
public class MyNamespaceFilter implements javax.xml.stream.StreamFilter {
public boolean accept(XMLStreamReader reader) {
    // Only interested in START_ELEMENT Events
    if (!reader.isStartElement()) { return false; }
    // Only interested in my desired Namespace
    String startElementNamespace = reader.getNamespaceURI();
    if (startElementNamespace == null
        || !startElementNamespace.equals(myDesiredNamespace)) {
        return false;
    }
    // of interest
    return true;
}
```

StreamFilter Example #2 Results

```
<BookCatalogue
xmlns="http://ignore"
xmlns:accept="
  http://accept">
<accept:Book ...>
<Book ...>
<Book ...>
...
```

```
<Event>
  [java]
  START_ELEMENT(1)
  [java] Name:
  {http://accept}Book
  [java] Attribute:
  {}:ISBN(CDATA)=81-
  40-34319-4
  [java] </Event>
  {... rest ignored ...}
```



Choosing between Cursor and Iterator APIs

Why 2 APIs?

- Given these wide-ranging development categories (see next slide), the StAX authors felt it was more useful to define two small, efficient APIs rather than overloading one larger and necessarily more complex API.

Developer Types

- Library and infrastructure developers:
 - Create application servers, JAXM, JAXB, JAX-RPC and similar implementations; need highly efficient, low-level APIs with minimal extensibility requirements.
- J2ME developers:
 - Need small, simple, pull-parsing libraries, and have minimal extensibility needs.
- Java EE and Java SE developers:
 - Need clean, efficient pull-parsing libraries, plus need the flexibility to both read and write XML streams, create new event types, and extend XML document elements and attributes.

General Recommendations

- If you are programming for a particularly memory-constrained environment, like J2ME, you can make smaller, more efficient code with the cursor API.
- If performance is your highest priority--for example, when creating low-level libraries or infrastructure--the cursor API is more efficient.
- If you want to create XML processing pipelines, use the iterator API.

General Recommendations

- If you want to modify the event stream, use the iterator API.
- If you want your application to be able to handle pluggable processing of the event stream, use the iterator API.
- In general, if you do not have a strong preference one way or the other, using the iterator API is recommended because it is more flexible and extensible, thereby "future-proofing" your applications.



Thank You!

