**Streaming API for XML (StAX)**

Table of Contents

[1.Introduction 2](#__RefHeading__1227_1044315592)

[2.Streaming Model Vs Document Object Model 2](#__RefHeading__1229_1044315592)

[3.Pull parsing Vs Push parsing 2](#__RefHeading__1231_1044315592)

[4.StAX Use Cases 4](#__RefHeading__1233_1044315592)

[5.Comparing StAX with other JAXP APIs 4](#__RefHeading__1235_1044315592)

[6.StAX API 13](#__RefHeading__1237_1044315592)

[6.1Cursor API (Forward only) 13](#__RefHeading__1239_1044315592)

[6.2Event Iterator API 14](#__RefHeading__1241_1044315592)

[7.StAX Factory Classes 17](#__RefHeading__1243_1044315592)

[8.Appendix 19](#__RefHeading__1245_1044315592)

[8.1Java Specification Requirement 19](#__RefHeading__1247_1044315592)

[8.2Examples 19](#__RefHeading__1249_1044315592)

# Introduction

Streaming API for XML (StAX) is a streaming, event-driven, pull parsing API for reading and writing XML documents. It enables us to create bidirectional XML parsers (it can both read and write XML documents) that are fast, easy to program, and have a light weight memory footprint.

StAX is the latest API in the JAXP family, an alternative to SAX, DOM, and TrAX, provides simpler programming model than SAX and more efficient memory management than DOM.

The primary goal of the StAX API is to give “parsing control to the programmer by exposing a simple iterator based API. This allows the programmer to ask for the next event (pull the event) and allows state to be stored in procedural fashion.”

# Streaming Model Vs Document Object Model

The DOM model involves creating in-memory objects representing an entire document tree and the complete infoset state for an XML document. Once in memory, DOM trees can be navigated freely and parsed arbitrarily but requires huge memory and increased processor requirements. This may not be an issue when working with small documents.

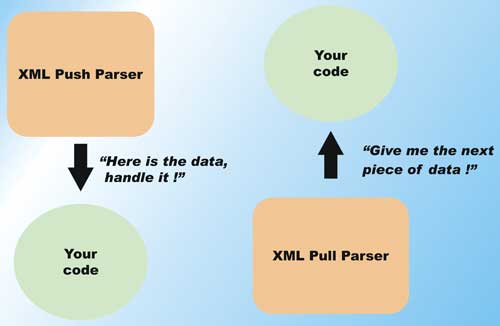
Streaming refers to a programming model in which XML infosets are transmitted and parsed serially at runtime. Stream-based parsers can start generating output immediately and the infosets can be discarded and garbage collected immediately after they are used. It provides a smaller memory footprint, reduced processor requirements and higher performance in certain situations.

# Pull parsing Vs Push parsing

Streaming 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 i.e. the client only gets (pulls) XML data when it explicitly asks for it.

Streaming 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 i.e. the parser sends the data whether or not the client is ready to use it at that time.

**Image 3.1** – Push Vs Pull



**Table 3.1** - Advantages of Pull parsing over Push parsing

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Pull parsing** | **Push parsing** |
| 1 | Client controls the application thread, and can call methods on the parser when needed. | Parser controls the application thread, and the client can only accept the invocation from the parser. |
| 2 | Client can read multiple documents at a time with a single thread. | NA |
| 3 | Supports XML views of non-XML data | NA |
| 4 | Parser library can be much smaller and the client code to interact with those libraries can also be simpler. | Parser library will be bigger. |
| 5 | Parser can filter XML documents such that elements unnecessary to the client can be ignored. | NA |

# StAX Use Cases

* Data Binding
* Marshalling an XML document
* Unmarshalling an XML document
* Parallel document processing
* Wireless communication
* SOAP message processing (Axis2 uses StAX model)
* Parsing WSDL
* Parsing graph representations with forward references
* Parsing simple predictable structures
* Virtual data sources
* Viewing as XML data stored in databases
* Navigating a DOM tree as a stream of events
* Viewing data in Java objects created by XML data binding
* Parsing specific XML vocabularies
* Pipelined XML processing

# Comparing StAX with other JAXP APIs

**DOM Interface**

* Loads entire XML document into a runtime memory at once.
* Provides large memory footprint.
* Allows you to read, create, update, and delete XML document.
* The DOM parser is called a DocumentBuilder, as it builds an in-memory Document representation, created by the DocumentBuilderFactory.

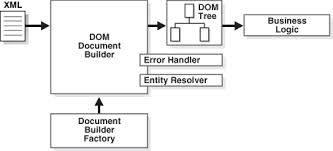
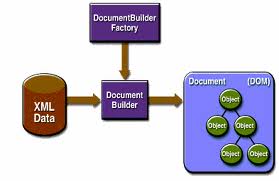
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();

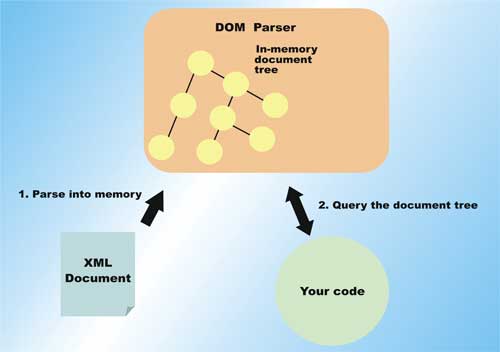
Document doc = builder.newDocument(); // To create new doc

Document doc = builder.parse(“filePath”); // To read

**Image 5.1** – DOM Architecture



**Image 5.2** – DOM Operation



**SAX Interface**

* Sends/Pushes events to the user defined event handler (DefaultHandler) by invoking the call-back methods (startDocument, endDocument, startElement, endElement, characters, etc.,)
* Read-only API, one cannot use it to build or update XML document.
* Provides small memory footprint and faster than DOM.
* The SAX parser (SAXParser) is created by SAXParserFactory.

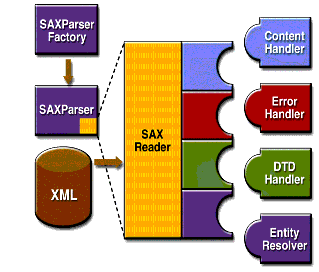
SAXParserFactory spf = SAXParserFactory.newInstance();

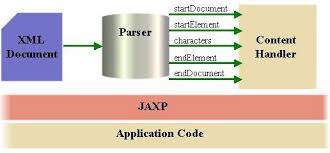
SAXParser saxParser = spf.newSAXParser();

XMLReader xmlReader = saxParser.getXMLReader();

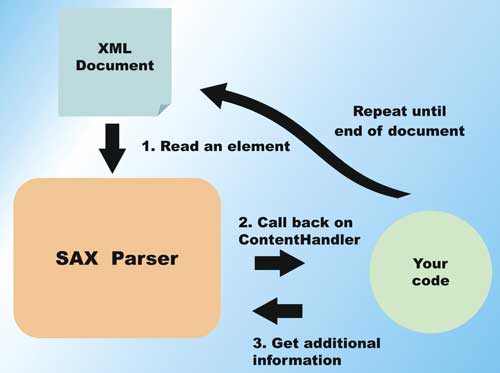
xmlReader.setContentHandler(new MyContentHandler());

**Image 5.3** – SAX Architecture





**Image 5.4** – SAX Operation



**XSLT Interface (TrAX)**

* The XML Stylesheet Language Transformations (**XSLT**) is a language for transforming XML documents into other XML documents or other object data (HTML, Plain Text, Web pages, XSL-FO, etc,).
* **XSLT processor** takes one or more XML documents, plus on or more XSLT stylesheet modules, and processes them to produce an output document.
* TrAX uses XSLT transformation for conversion of XML document into other forms of data.
* JAXP provides interfaces in javax.xml.transform package to allow application to invoke an XSLT transformation.
* The TrAX parser or XSLT processor is created by **TranformerFactory**.
* The objects of **Templates**, representing the compiled form of a stylesheet, created by methods on a TransformerFactory, are thread-safe and can be used repeatedly to apply style sheet to multiple source XML documents.
* The **Transformer**, representing the executable form of stylesheet, is shared across the threads and provides methods to set serialization options and run the actual transformation.
* The abstract interfaces Source and Result are defined to represent Input and Output of the transformation, respectively.

TransformerFactory factory = TransformerFactory.newInstance();

// Input

Source src = new StreamSource(new FileInputStream(new File("XMLFilePath.xml")););

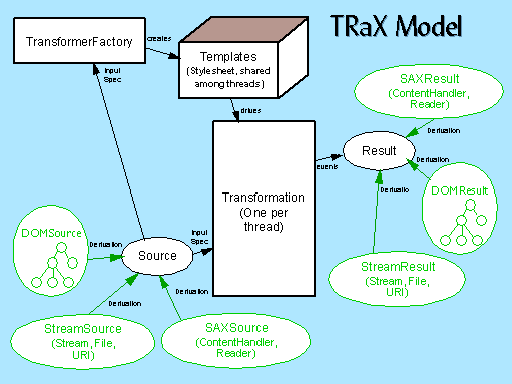
// Output

Result res = new StAXResult(xmlEventWriter);

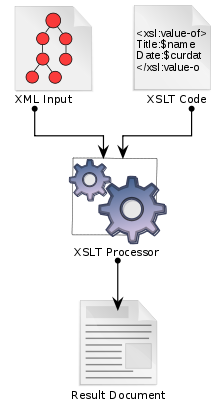
Transformer transformer = factory.newTransformer(new StreamSource(new FileInputStream(new File("XSLFilePath"))));

transformer.transform(src, res);

**Image 5.5** – TrAX Architecture



**Image 5.6** – XSLT Processing



**StAX Interface**

Refer Introduction section (**§**1) which briefs about StAX, and for more details go through upcoming sections.

**Image 5.7** – StAX Operation



**Comparison**

* StAX can be compared to SAX since StAX is not as powerful or flexible as DOM or TrAX.

**StAX Vs SAX**

* StAX clients are easier to code than SAX clients.
* StAX is bidirectional API; it can both read and write XML documents, but SAX is read only API.
* StAX is a pull API, whereas SAX is a push API.

**Table 4.1 –** XML Parser API Feature Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 | Not supported | Not supported | Supported | Supported |
| CPU and Memory Efficiency | Good | Good | Varies | Varies |
| Forward Only | Supported | Supported | Not supported | Not supported |
| Read XML | Supported | Supported | Supported | Supported |
| Write XML | Supported | Not supported | Supported | Supported |
| Create, Read, Update, Delete | Not supported | Not supported | Supported | Not supported |

# StAX API

The StAX API is two distinct API sets: a cursor API and an iterator API.

# Cursor API (Forward only)

The StAX Cursor API represents a cursor with which you can walk an XML document from beginning to end. This cursor can point to one infoset element at a time, and always moves forward, never backward.

The two main cursor interfaces are,

1. XMLStreamReader

The XMLStreamReader includes accessor methods for retrieving information from XML, including:

* + - Document encoding
    - Element name
    - Attribute
    - Namespace
    - Text node
    - Start tag
    - Comment
    - Processing element
    - Document boundary
    - Etc.,

You can call methods on XMLStreamReader, such as getText and getName, to get data at the current cursor location.

public interface XMLStreamReader {

public int next() throws XMLStreamException;

public boolean hasNext() throws XMLStreamException;

public String getText();

public String getLocalName();

public String getNamespaceURI();

// ... other methods not shown

}

1. XMLStreamWriter

XMLStreamWriter provides methods that correspond to StartElement and EndElement event types.

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

}

# Event Iterator API

The StAX Event Iterator API represents an XML document as a set of discrete event objects. These event objects are pulled by the application and provided by the parser in the order in which they are read in the source XML document.

The two primary iterator interfaces for reading and writing events are,

1. XMLEventReader

It has five methods, the most important of which is nextEvent, which returns the next event in an XML stream.

And it implements java.util.Iterator, which means that the returns from XMLEventReader can be cached or passed into other methods.

public interface XMLEventReader extends Iterator {

public XMLEvent nextEvent() throws XMLStreamException;

public boolean hasNext();

public XMLEvent peek() throws XMLStreamException;

public String getElementText() throws XMLStreamException;

// ... other methods not shown

}

1. XMLEventWriter

It has methods to write event objects into a document.

public interface XMLEventWriter extends XMLEventConsumer {

public void flush() throws XMLStreamException;

public void close() throws XMLStreamException;

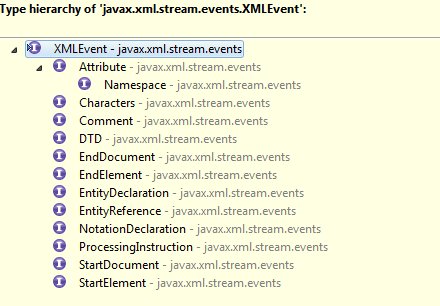
public void add(XMLEvent e) throws XMLStreamException;

public void add(Attribute attribute) throws XMLStreamException;

// ... other methods not shown

}

The base iterator event interface is called XMLEvent and there are sub interfaces for each event type.



**Notes:**

* The events are created in **the order** in which the corresponding XML elements are encountered in the document, including nesting of elements, opening and closing of elements, attribute order, document start and document end, and so forth.
* As with **proper XML syntax**, all container elements have corresponding start and end events; for example, every StartElement has a corresponding EndElement, even for empty elements.
* Attribute events are treated as **secondary events**, and are accessed from their corresponding StartElement event.
* Similar to Attribute events, **Namespace events** are treated as secondary, but appear twice and are accessible twice in the event stream, first from their corresponding StartElement and then from their corresponding EndElement.
* **Character events** are specified for all elements, even if those elements have no character data. Similarly, Character events can be split across events.
* The StAX parser maintains a **namespace stack**, which holds information about all XML namespaces defined for the current element and its ancestors. The namespace stack, which is exposed through the javax.xml.namespace.NamespaceContext interface, can be accessed by namespace prefix or URI.

**Choosing between Cursor API and Iterator API**

* Objects created through the XMLEvent subclasses are immutable, and can be used in arrays and collections, and can be passed to other application routines even after the parser has moved on to the next subsequent events.
* You can create new user defined events by implementing XMLEvent.
* You can add or remove events from an XML stream in much simpler ways than with the cursor API.

Prefer Cursor API when,

1. Programming for a **memory-constrained** application, like J2ME.
2. **Performance** is high priority.

Prefer Iterator API when,

1. You want to create **XML processing pipelines**.
2. You want your application to be able to handle **pluggable processing** of the event streams.
3. You want to **modify the event stream**.
4. In general, you **don’t have a strong preference** on what to use.

# StAX Factory Classes

The StAX factory classes let you define and configure implementation instances of XML stream reader, stream writer and event classes.

The StAX programmers can create XML stream readers, stream writers, and events by using XMLInputFactory, XMLoutputFactory, and XMLEventFactory classes.

**StAX factory implementation class lookup procedures**

Deriving from JAXP, the StAX factories’ newInstance method determines the specific implementation class to load by using the following procedures:

1. Use the system property *javax.xml.stream.XMLInputFactory*
2. Use the lib/*javax.xml.stream.properties* file in JRE directory.
3. Use the Services API, looking at the *META-INF/services/javax.xml.stream.XMLInputFactory* files in JAR files available to the JRE.
4. Use the *platform default* StAX factory
   1. **XMLInputFactory Class**

The XMLInputFactory class lets you configure implementation instances of XML stream reader processors created by the factory.

New factory instances of an abstract class XMLInputFactory are created by calling a static method newInstance on the class.

XMLInputFactory readerFactory = XMLInputFactory.newInstance();

// Cursor API

readerFactory.createXMLStreamReader(new FileReader(new File(filePath)));

// Iterator API

readerFactory.createXMLEventReader(new FileReader(new File(filePath)));

* 1. **XMLOutputFactory Class**

The XMLOutputFactory class lets you configure implementation instances of XML stream writer processors created by the factory.

New factory instances of an abstract class XMLOutputFactory are created by calling a static method newInstance on the class.

XMLOutputFactory writerFactory = XMLOutputFactory.newInstance();

// Cursor API

writerFactory.createXMLStreamWriter(new FileWriter(new File(filePath)));

// Iterator API

writerFactory.createXMLEventWriter(new FileWriter(new File(filePath)));

* 1. **XMLEventFactory Class**

The XMLEventFactory class lets you configure implementation instances of XML events created by the factory.

New factory instances of an abstract class XMLEventFactory are created by calling a static method newInstance on the class.

XMLEventFactory eventFactory = XMLEventFactory.newInstance();

* 1. **XMLReporter, XMLResolver and XMLEventAllocator**

**XMLReporter**

All non-fatal errors and warnings are reported using the XMLReporter interface.

All fatal errors are reported by XMLStreamException.

**XMLResolver**

The XMLResolver interface provides a means to set the method that resolves resources during XML processing. An application sets the interface on XMLInputFactory, which then sets the interface on all processors created by that factory instance.

**XMLEventAllocator**

XMLEventAllocator can be used to get event information as an XMLEvent object, even when using the cursor API.

# Appendix

# Java Specification Requirement

Find below the Streaming API for XML (JSR-173) for JavaTM Specification by BEA (**B**ill Coleman, **E**d Scott and **A**lfred Chuang) Systems, Inc.

# Examples