

◆ ◆ ◆ 18 CHAPTER 18

Streaming API for XML

This chapter focuses on the Streaming API for XML (StAX), a streaming Java-based, event-driven, pull-parsing API 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, DOM, TrAX, and DOM for developers looking to do high-performance stream filtering, processing, and modification, particularly with low memory and limited extensibility requirements.

To summarize, StAX provides a standard, bidirectional *pull parser* interface for streaming XML processing, offering a simpler programming model than SAX and more efficient memory management than DOM. StAX enables developers to parse and modify XML streams as events, and to extend XML information models to allow application-specific additions. More detailed comparisons of StAX with several alternative APIs are provided below, in “[Comparing StAX to Other JAXP APIs](#)” on page 539.

Why StAX?

The StAX project was spearheaded by BEA with support from Sun Microsystems, and the JSR 173 specification passed the Java Community Process final approval ballot in March, 2004 (<http://jcp.org/en/jsr/detail?id=173>). 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.” StAX was created to address limitations in the two most prevalent parsing APIs, SAX and DOM.

Streaming versus DOM

Generally speaking, there are two programming models for working with XML infosets: *streaming* and the *document object model* (DOM).

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, and as such provide maximum flexibility for developers. However, the cost of this flexibility is a potentially large memory footprint and significant processor requirements, because the entire representation of the document must be held in memory as objects for the duration of the document processing. This may not be an issue when working with small documents, but memory and processor requirements can escalate quickly with document size.

Streaming refers to a programming model in which XML infosets are transmitted and parsed serially at application runtime, often in real time, and often from dynamic sources whose contents are not precisely known beforehand. Moreover, stream-based parsers can start generating output immediately, and infoset elements can be discarded and garbage collected immediately after they are used. While providing a smaller memory footprint, reduced processor requirements, and higher performance in certain situations, the primary trade-off with stream processing is that you can only see the infoset state at one location at a time in the document. You are essentially limited to the “cardboard tube” view of a document, the implication being that you need to know what processing you want to do before reading the XML document.

Streaming models for XML processing are particularly useful when your application has strict memory limitations, as with a cell phone running J2ME, or when your application needs to simultaneously process several requests, as with an application server. In fact, it can be argued that the majority of XML business logic can benefit from stream processing, and does not require the in-memory maintenance of entire DOM trees.

Pull Parsing versus 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; that is, 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; that is, the parser sends the data whether or not the client is ready to use it at that time.

Pull parsing provides several advantages over push parsing when working with XML streams:

- With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.
- Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries, even for more complex documents.
- 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, and it can support XML views of non-XML data.

StAX Use Cases

The StAX specification defines a number of use cases for the API:

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

A complete discussion of all these use cases is beyond the scope of this chapter. Please refer to the StAX specification for further information.

Comparing StAX to Other JAXP APIs

As an API in the JAXP family, StAX can be compared, among other APIs, to SAX, TrAX, and JDOM. Of the latter two, StAX is not as powerful or flexible as TrAX or JDOM, but neither does it require as much memory or processor load to be useful, and StAX can, in many cases, outperform the DOM-based APIs. The same arguments outlined above, weighing the cost/benefits of the DOM model versus the streaming model, apply here.

With this in mind, the closest comparisons can be made between StAX and SAX, and it is here that StAX offers features that are beneficial in many cases; some of these include:

- StAX-enabled clients are generally easier to code than SAX clients. While it can be argued that SAX parsers are marginally easier to write, StAX parser code can be smaller and the code necessary for the client to interact with the parser simpler.
- 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.
- SAX is a push API, whereas StAX is pull. The trade-offs between push and pull APIs outlined above apply here.

Table 18–1 summarizes the comparative features of StAX, SAX, DOM, and TrAX (table adapted from “Does StAX Belong in Your XML Toolbox?” at <http://www.developer.com/xml/article.php/3397691/Does-StAX-Belong-in-Your-XML-Toolbox.htm> by Jeff Ryan).

TABLE 18–1 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 exposes methods for iterative, event-based processing of XML documents. XML documents are treated as a filtered series of events, and infoset states can be stored in a procedural fashion. Moreover, unlike SAX, the StAX API is bidirectional, enabling both reading and writing of XML documents.

The StAX API is really two distinct API sets: a *cursor* API and an *iterator* API. These two API sets explained in greater detail later in this chapter, but their main features are briefly described below.

Cursor API

As the name implies, 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 thing at a time, and always moves forward, never backward, usually one info:et element at a time.

The two main cursor interfaces are `XMLStreamReader` and `XMLStreamWriter`. `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; for example:

```
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
}
```

You can call methods on `XMLStreamReader`, such as `getText` and `getName`, to get data at the current cursor location. `XMLStreamWriter` provides methods that correspond to `StartElement` and `EndElement` event types; for example:

```
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
}
```

The cursor API mirrors SAX in many ways. For example, methods are available for directly accessing string and character information, and integer indexes can be used to access attribute and namespace information. As with SAX, the cursor API methods return XML information as strings, which minimizes object allocation requirements.

Iterator API

The StAX *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.

The base iterator interface is called `XMLEvent`, and there are subinterfaces for each event type listed in [Table 18–2](#). The primary parser interface for reading iterator events is `XMLEventReader`, and the primary interface for writing iterator events is `XMLEventWriter`. The

`XMLStreamReader` interface contains five methods, the most important of which is `nextEvent`, which returns the next event in an XML stream. `XMLStreamReader` implements `java.util.Iterator`, which means that returns from `XMLStreamReader` can be cached or passed into routines that can work with the standard Java Iterator; for example:

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

Similarly, on the output side of the iterator API, you have:

```
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;
    ...
}
```

Iterator Event Types

[Table 18–2](#) lists the `XMLEvent` types defined in the event iterator API.

TABLE 18–2 `XMLEvent` Types

Event Type	Description
StartDocument	Reports the beginning of a set of XML events, including encoding, XML version, and standalone properties.
StartElement	Reports the start of an element, including any attributes and namespace declarations; also provides access to the prefix, namespace URI, and local name of the start tag.
EndElement	Reports the end tag of an element. Namespaces that have gone out of scope can be recalled here if they have been explicitly set on their corresponding <code>StartElement</code> .
Characters	Corresponds to XML CDATA sections and <code>CharacterData</code> entities. Note that ignorable white space and significant white space are also reported as <code>Character</code> events.
EntityReference	Character entities can be reported as discrete events, which an application developer can then choose to resolve or pass through unresolved. By default, entities are resolved. Alternatively, if you do not want to report the entity as an event, replacement text can be substituted and reported as <code>Characters</code> .
ProcessingInstruction	Reports the target and data for an underlying processing instruction.
Comment	Returns the text of a comment.

TABLE 18–2 XMLEvent Types (Continued)

Event Type	Description
EndDocument	Reports the end of a set of XML events.
DTD	Reports as <code>java.lang.String</code> information about the DTD, if any, associated with the stream, and provides a method for returning custom objects found in the DTD.
Attribute	Attributes are generally reported as part of a <code>StartElement</code> event. However, there are times when it is desirable to return an attribute as a standalone <code>Attribute</code> event; for example, when a namespace is returned as the result of an XQuery or XPath expression.
Namespace	As with attributes, namespaces are usually reported as part of a <code>StartElement</code> , but there are times when it is desirable to report a namespace as a discrete <code>Namespace</code> event.

Note that the DTD, `EntityDeclaration`, `EntityReference`, `NotationDeclaration`, and `ProcessingInstruction` events are only created if the document being processed contains a DTD.

Example of Event Mapping

As an example of how the event iterator API maps an XML stream, consider the following XML document:

```
<?xml version="1.0"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
    <Title>Yogasana Vijnana: the Science of Yoga</Title>
    <ISBN>81-40-34319-4</ISBN>
    <Cost currency="INR">11.50</Cost>
  </Book>
</BookCatalogue>
```

This document would be parsed into eighteen primary and secondary events, as shown in [Table 18–3](#). Note that secondary events, shown in curly braces (`{}`), are typically accessed from a primary event rather than directly.

TABLE 18–3 Example of Iterator API Event Mapping

#	Element/Attribute	Event
1	<code>version="1.0"</code>	<code>StartDocument</code>
2	<code>isCDATA = false</code> <code>data = "\n"</code> <code>isWhiteSpace = true</code>	<code>Characters</code>

TABLE 18-3 Example of Iterator API Event Mapping (Continued)

#	Element/Attribute	Event
3	qname = BookCatalogue:http://www.publishing.org attributes = null namespaces = {BookCatalogue" -> http://www.publishing.org"}	StartElement
4	qname = Book attributes = null namespaces = null	StartElement
5	qname = Title attributes = null namespaces = null	StartElement
6	isCDATA = false data = "Yogasana Vijnana: the Science of Yoga\n\t" isWhiteSpace = false	Characters
7	qname = Title namespaces = null	EndElement
8	qname = ISBN attributes = null namespaces = null	StartElement
9	isCDATA = false data = "81-40-34319-4\n\t" isWhiteSpace = false	Characters
10	qname = ISBN namespaces = null	EndElement
11	qname = Cost attributes = {"currency" -> INR} namespaces = null	StartElement
12	isCDATA = false data = "11.50\n\t" isWhiteSpace = false	Characters
13	qname = Cost namespaces = null	EndElement
14	isCDATA = false data = "\n" isWhiteSpace = true	Characters
15	qname = Book namespaces = null	EndElement

TABLE 18-3 Example of Iterator API Event Mapping (Continued)

#	Element/Attribute	Event
16	isCDATA = false data = "\n" isWhiteSpace = true	Characters
17	qname = BookCatalogue:http://www.publishing.org namespaces = {BookCatalogue" -> http://www.publishing.org"}	EndElement
18		EndDocument

There are several important things to note in this example:

- 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 and Iterator APIs

It is reasonable to ask at this point, “What API should I choose? Should I create instances of `XMLStreamReader` or `XMLEventReader`? Why are there two kinds of APIs anyway?”

Development Goals

The authors of the StAX specification targeted three types of developers:

- **Library and infrastructure developers:** 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.

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

Comparing Cursor and Iterator APIs

Before choosing between the cursor and iterator APIs, you should note a few things that you can do with the iterator API that you cannot do with cursor API:

- Objects created from the `XMLEvent` subclasses are immutable, and can be used in arrays, lists, and maps, and can be passed through your applications even after the parser has moved on to subsequent events.
- You can create subtypes of `XMLEvent` that are either completely new information items or extensions of existing items but with additional methods.
- You can add and remove events from an XML event stream in much simpler ways than with the cursor API.

Similarly, keep some general recommendations in mind when making your choice:

- 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.
- 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.

Using StAX

In general, StAX programmers create XML stream readers, writers, and events by using the `XMLInputFactory`, `XMLOutputFactory`, and `XMLEventFactory` classes. Configuration is done by setting properties on the factories, whereby implementation-specific settings can be passed to the underlying implementation using the `setProperty` method on the factories. Similarly, implementation-specific settings can be queried using the `getProperty` factory method.

The `XMLInputFactory`, `XMLOutputFactory`, and `XMLEventFactory` classes are described below, followed by discussions of resource allocation, namespace and attribute management, error handling, and then finally reading and writing streams using the cursor and iterator APIs.

StAX Factory Classes

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

XMLInputFactory Class

The `XMLInputFactory` class lets you configure implementation instances of XML stream reader processors created by the factory. New instances of the abstract class `XMLInputFactory` are created by calling the `newInstance` method on the class. The static method `XMLInputFactory.newInstance` is then used to create a new factory instance.

Deriving from JAXP, the `XMLInputFactory.newInstance` method determines the specific `XMLInputFactory` implementation class to load by using the following lookup procedure:

1. Use the `javax.xml.stream.XMLInputFactory` system property.
2. Use the `lib/xml.stream.properties` file in the J2SE Java Runtime Environment (JRE) directory.
3. Use the Services API, if available, to determine the classname by looking in the `META-INF/services/javax.xml.stream.XMLInputFactory` files in JAR files available to the JRE.
4. Use the platform default `XMLInputFactory` instance.

After getting a reference to an appropriate `XMLInputFactory`, an application can use the factory to configure and create stream instances. [Table 18–4](#) lists the properties supported by `XMLInputFactory`. See the StAX specification for a more detailed listing.

TABLE 18-4 `javax.xml.stream.XMLInputFactory` Properties

Property	Description
<code>isValidating</code>	Turns on implementation-specific validation.
<code>isCoalescing</code>	<i>(Required)</i> Requires the processor to coalesce adjacent character data.
<code>isNamespaceAware</code>	Turns off namespace support. All implementations must support namespaces. Support for non-namespace-aware documents is optional.
<code>isReplacingEntityReferences</code>	<i>(Required)</i> Requires the processor to replace internal entity references with their replacement value and report them as characters or the set of events that describe the entity.
<code>isSupportingExternalEntities</code>	<i>(Required)</i> Requires the processor to resolve external parsed entities.
<code>reporter</code>	<i>(Required)</i> Sets and gets the implementation of the <code>XMLReporter</code> interface.
<code>resolver</code>	<i>(Required)</i> Sets and gets the implementation of the <code>XMLResolver</code> interface.
<code>allocator</code>	<i>(Required)</i> Sets and gets the implementation of the <code>XMLEventAllocator</code> interface.

XMLOutputFactory Class

New instances of the abstract class `XMLOutputFactory` are created by calling the `newInstance` method on the class. The static method `XMLOutputFactory.newInstance` is then used to create a new factory instance. The algorithm used to obtain the instance is the same as for `XMLInputFactory` but references the `javax.xml.stream.XMLOutputFactory` system property.

`XMLOutputFactory` supports only one property, `javax.xml.stream.isRepairingNamespaces`. This property is required, and its purpose is to create default prefixes and associate them with Namespace URIs. See the StAX specification for more information.

XMLEventFactory Class

New instances of the abstract class `XMLEventFactory` are created by calling the `newInstance` method on the class. The static method `XMLEventFactory.newInstance` is then used to create a new factory instance. This factory references the `javax.xml.stream.XMLEventFactory` property to instantiate the factory. The algorithm used to obtain the instance is the same as for `XMLInputFactory` and `XMLOutputFactory` but references the `javax.xml.stream.XMLEventFactory` system property.

There are no default properties for `XMLEventFactory`.

Resources, Namespaces, and Errors

The StAX specification handles resource resolution, attributes and namespace, and errors and exceptions as described below.

Resource Resolution

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.

Attributes and Namespaces

Attributes are reported by a StAX processor using lookup methods and strings in the cursor interface, and `Attribute` and `Namespace` events in the iterator interface. Note here that namespaces are treated as attributes, although namespaces are reported separately from attributes in both the cursor and iterator APIs. Note also that namespace processing is optional for StAX processors. See the StAX specification for complete information about namespace binding and optional namespace processing.

Error Reporting and Exception Handling

All fatal errors are reported by way of the `javax.xml.stream.XMLStreamException` interface. All nonfatal errors and warnings are reported using the `javax.xml.stream.XMLReporter` interface.

Reading XML Streams

As described earlier in this chapter, the way you read XML streams with a StAX processor, and what you get back, vary significantly depending on whether you are using the StAX cursor API or the event iterator API. The following two sections describe how to read XML streams with each of these APIs.

Using XMLStreamReader

The `XMLStreamReader` interface in the StAX cursor API lets you read XML streams or documents in a forward direction only, one item in the infoset at a time. The following methods are available for pulling data from the stream or skipping unwanted events:

- Get the value of an attribute
- Read XML content
- Determine whether an element has content or is empty
- Get indexed access to a collection of attributes
- Get indexed access to a collection of namespaces

- Get the name of the current event (if applicable)
- Get the content of the current event (if applicable)

Instances of `XMLStreamReader` have at any one time a single current event on which its methods operate. When you create an instance of `XMLStreamReader` on a stream, the initial current event is the `START_DOCUMENT` state. The `XMLStreamReader.next` method can then be used to step to the next event in the stream.

Reading Properties, Attributes, and Namespaces

The `XMLStreamReader.next` method loads the properties of the next event in the stream. You can then access those properties by calling the `XMLStreamReader.getLocalName` and `XMLStreamReader.getText` methods.

When the `XMLStreamReader` cursor is over a `StartElement` event, it reads the name and any attributes for the event, including the namespace. All attributes for an event can be accessed using an index value, and can also be looked up by namespace URI and local name. Note, however, that only the namespaces declared on the current `StartElement` are available; previously declared namespaces are not maintained, and redeclared namespaces are not removed.

XMLStreamReader Methods

`XMLStreamReader` provides the following methods for retrieving information about namespaces and attributes:

```
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);
```

Namespaces can also be accessed using three additional methods:

```
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);
```

Instantiating an XMLStreamReader

This example, taken from the StAX specification, shows how to instantiate an input factory, create a reader, and iterate over the elements of an XML stream:

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
```

```
while(r.hasNext()) {
    r.next();
}
```

Using XMLEventReader

The XMLEventReader API in the StAX event iterator API provides the means to map events in an XML stream to allocated event objects that can be freely reused, and the API itself can be extended to handle custom events.

XMLEventReader provides four methods for iteratively parsing XML streams:

- next: Returns the next event in the stream
- nextEvent: Returns the next typed XMLEvent
- hasNext: Returns true if there are more events to process in the stream
- peek: Returns the event but does not iterate to the next event

For example, the following code snippet illustrates the XMLEventReader method declarations:

```
package javax.xml.stream;
import java.util.Iterator;
public interface XMLEventReader extends Iterator {
    public Object next();
    public XMLEvent nextEvent() throws XMLStreamException;
    public boolean hasNext();
    public XMLEvent peek() throws XMLStreamException;
    ...
}
```

To read all events on a stream and then print them, you could use the following:

```
while(stream.hasNext()) {
    XMLEvent event = stream.nextEvent();
    System.out.print(event);
}
```

Reading Attributes

You can access attributes from their associated `javax.xml.stream.StartElement`, as follows:

```
public interface StartElement extends XMLEvent {
    public Attribute getAttributeByName(QName name);
    public Iterator getAttributes();
}
```

You can use the `getAttributes` method on the `StartElement` interface to use an `Iterator` over all the attributes declared on that `StartElement`.

Reading Namespaces

Similar to reading attributes, namespaces are read using an `Iterator` created by calling the `getNamespaces` method on the `StartElement` interface. Only the namespace for the current `StartElement` is returned, and an application can get the current namespace context by using `StartElement.getNamespaceContext`.

Writing XML Streams

StAX is a bidirectional API, and both the cursor and event iterator APIs have their own set of interfaces for writing XML streams. As with the interfaces for reading streams, there are significant differences between the writer APIs for cursor and event iterator. The following sections describe how to write XML streams using each of these APIs.

Using XMLStreamWriter

The `XMLStreamWriter` interface in the StAX cursor API lets applications write back to an XML stream or create entirely new streams. `XMLStreamWriter` has methods that let you:

- Write well-formed XML
- Flush or close the output
- Write qualified names

Note that `XMLStreamWriter` implementations are not required to perform well-formedness or validity checks on input. While some implementations may perform strict error checking, others may not. The rules you implement are applied to properties defined in the `XMLOutputFactory` class.

The `writeCharacters` method is used to escape characters such as `&`, `<`, `>`, and `"`. Binding prefixes can be handled by either passing the actual value for the prefix, by using the `setPrefix` method, or by setting the property for defaulting namespace declarations.

The following example, taken from the StAX specification, shows how to instantiate an output factory, create a writer, and write XML output:

```
XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLStreamWriter writer = output.createXMLStreamWriter( ... );
writer.writeStartDocument();
writer.setPrefix("c", "http://c");
writer.setDefaultNamespace("http://c");
writer.writeStartElement("http://c", "a");
writer.writeAttribute("b", "blah");
writer.writeNamespace("c", "http://c");
writer.writeDefaultNamespace("http://c");
writer.setPrefix("d", "http://c");
writer.writeEmptyElement("http://c", "d");
writer.writeAttribute("http://c", "chris", "fry");
writer.writeNamespace("d", "http://c");
```



```
writer.writeCharacters("Jean Arp");
writer.writeEndElement();
writer.flush();
```

This code generates the following XML (new lines are non-normative):

```
<?xml version='1.0' encoding='utf-8'?>
<a b="blah" xmlns:c="http://c" xmlns="http://c">
<d:d d:chris="fry" xmlns:d="http://c"/>Jean Arp</a>
```

Using XMLEventWriter

The `XMLEventWriter` interface in the StAX event iterator API lets applications write back to an XML stream or create entirely new streams. This API can be extended, but the main API is as follows:

```
public interface XMLEventWriter {
    public void flush() throws XMLStreamException;
    public void close() throws XMLStreamException;
    public void add(XMLEvent e) throws XMLStreamException;
    // ... other methods not shown.
}
```

Instances of `XMLEventWriter` are created by an instance of `XMLOutputFactory`. Stream events are added iteratively, and an event cannot be modified after it has been added to an event writer instance.

Attributes, Escaping Characters, Binding Prefixes

StAX implementations are required to buffer the last `StartElement` until an event other than `Attribute` or `Namespace` is added or encountered in the stream. This means that when you add an `Attribute` or a `Namespace` to a stream, it is appended the current `StartElement` event.

You can use the `Characters` method to escape characters like `&`, `<`, `>`, and `"`.

The `setPrefix(...)` method can be used to explicitly bind a prefix for use during output, and the `getPrefix(...)` method can be used to get the current prefix. Note that by default, `XMLEventWriter` adds namespace bindings to its internal namespace map. Prefixes go out of scope after the corresponding `EndElement` for the event in which they are bound.

Sun's Streaming XML Parser Implementation

Application Server includes Sun Microsystems' JSR 173 (StAX) implementation, called the Sun Java Streaming XML Parser (referred to as Streaming XML Parser). The Streaming XML Parser is a high-speed, non-validating, W3C XML 1.0 and Namespace 1.0-compliant streaming XML pull parser built upon the Xerces2 codebase.

In Sun's Streaming XML Parser implementation, the Xerces2 lower layers, particularly the Scanner and related classes, have been redesigned to behave in a pull fashion. In addition to the changes in the lower layers, the Streaming XML Parser includes additional StAX-related functionality and many performance-enhancing improvements. The Streaming XML Parser is implemented in the `appserv-ws.jar` and `javaee.jar` files, both of which are located in the `as-install/lib/` directory.

Included with this Java EE tutorial are StAX code examples, located in the `tut-install/javaeetutorial5/examples/stax/` directory, that illustrate how Sun's Streaming XML Parser implementation works. These examples are described in [“Example Code” on page 555](#).

Before you proceed with the example code, there are two aspects of the Streaming XML Parser of which you should be aware:

- [“Reporting CDATA Events” on page 554](#)
- [“Streaming XML Parser Factories Implementation” on page 554](#)

These topics are discussed below.

Reporting CDATA Events

The `javax.xml.stream.XMLStreamReader` implemented in the Streaming XML Parser does not report CDATA events. If you have an application that needs to receive such events, configure the `XMLInputFactory` to set the following implementation-specific `report-cdata-event` property:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty("report-cdata-event", Boolean.TRUE);
```

Streaming XML Parser Factories Implementation

Most applications do not need to know the factory implementation class name. Just adding the `javaee.jar` and `appserv-ws.jar` files to the classpath is sufficient for most applications because these two jars supply the factory implementation classname for various Streaming XML Parser properties under the `META-INF/services/` directory (for example, `javax.xml.stream.XMLInputFactory`, `javax.xml.stream.XMLOutputFactory`, and `javax.xml.stream.XMLEventFactory`).

However, there may be scenarios when an application would like to know about the factory implementation class name and set the property explicitly. These scenarios could include cases where there are multiple JSR 173 implementations in the classpath and the application wants to choose one, perhaps one that has superior performance, contains a crucial bug fix, or suchlike.

If an application sets the `SystemProperty`, it is the first step in a lookup operation, and so obtaining the factory instance would be fast compared to other options; for example:

```
javax.xml.stream.XMLInputFactory -->
    com.sun.xml.stream.ZephyrParserFactory
javax.xml.stream.XMLOutputFactory -->
    com.sun.xml.stream.ZephyrWriterFactory
javax.xml.stream.XMLEventFactory -->
    com.sun.xml.stream.events.ZephyrEventFactory
```

Example Code

This section steps through the example StAX code included in the Java EE 5 Tutorial bundle. All example directories used in this section are located in the *tut-install/javaeetutorial5/examples/stax/* directory.

The topics covered in this section are as follows:

- “Example Code Organization” on page 555
- “Example XML Document” on page 556
- “Cursor Example” on page 556
- “Cursor-to-Event Example” on page 558
- “Event Example” on page 560
- “Filter Example” on page 563
- “Read-and-Write Example” on page 565
- “Writer Example” on page 567

Example Code Organization

The *tut-install/javaeetutorial5/examples/stax/* directory contains the six StAX example directories:

- **Cursor example:** The cursor directory contains `CursorParse.java`, which illustrates how to use the `XMLStreamReader` (cursor) API to read an XML file.
- **Cursor-to-Event example:** The `cursor2event` directory contains `CursorApproachEventObject.java`, which illustrates how an application can get information as an `XMLEvent` object when using cursor API.
- **Event example:** The event directory contains `EventParse.java`, which illustrates how to use the `XMLEventReader` (event iterator) API to read an XML file.
- **Filter example:** The filter directory contains `MyStreamFilter.java`, which illustrates how to use the StAX Stream Filter APIs. In this example, the filter accepts only `StartElement` and `EndElement` events, and filters out the remainder of the events.
- **Read-and-Write example:** The `readnwrite` directory contains `EventProducerConsumer.java`, which illustrates how the StAX producer/consumer mechanism can be used to simultaneously read and write XML streams.
- **Writer example:** The writer directory contains `CursorWriter.java`, which illustrates how to use `XMLStreamWriter` to write an XML file programmatically.

All of the StAX examples except for the Writer example use an example XML document, `BookCatalog.xml`.

Example XML Document

The example XML document, `BookCatalog.xml`, used by most of the StAX example classes, is a simple book catalog based on the common `BookCatalogue` namespace. The contents of `BookCatalog.xml` are listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<BookCatalogue xmlns="http://www.publishing.org">
  <Book>
    <Title>Yogasana Vijnana: the Science of Yoga</Title>
    <author>Dhirendra Brahmachari</Author>
    <Date>1966</Date>
    <ISBN>81-40-34319-4</ISBN>
    <Publisher>Dhirendra Yoga Publications</Publisher>
    <Cost currency="INR">11.50</Cost>
  </Book>
  <Book>
    <Title>The First and Last Freedom</Title>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-064831-7</ISBN>
    <Publisher>Harper & Row</Publisher>
    <Cost currency="USD">2.95</Cost>
  </Book>
</BookCatalogue>
```

Cursor Example

Located in the `tut-install/javaeetutorial5/examples/stax/cursor/` directory, `CursorParse.java` demonstrates using the StAX cursor API to read an XML document. In the Cursor example, the application instructs the parser to read the next event in the XML input stream by calling `<code>next()</code>`.

Note that `<code>next()</code>` just returns an integer constant corresponding to underlying event where the parser is positioned. The application needs to call the relevant function to get more information related to the underlying event.

You can imagine this approach as a virtual cursor moving across the XML input stream. There are various accessor methods which can be called when that virtual cursor is at a particular event.

Stepping through Events

In this example, the client application pulls the next event in the XML stream by calling the `next` method on the parser; for example:

```

try {
    for (int i = 0 ; i < count ; i++) {
        // pass the file name.. all relative entity
        // references will be resolved against this as
        // base URI.
        XMLStreamReader xmlr =
            xmlif.createXMLStreamReader(filename,
                new FileInputStream(filename));
        // when XMLStreamReader is created, it is positioned
        // at START_DOCUMENT event.
        int eventType = xmlr.getEventType();
        printEventType(eventType);
        printStartDocument(xmlr);
        // check if there are more events in the input stream
        while(xmlr.hasNext()) {
            eventType = xmlr.next();
            printEventType(eventType);
            // these functions print the information about
            // the particular event by calling the relevant
            // function
            printStartElement(xmlr);
            printEndElement(xmlr);
            printText(xmlr);
            printPIData(xmlr);
            printComment(xmlr);
        }
    }
}

```

Note that next just returns an integer constant corresponding to the event underlying the current cursor location. The application calls the relevant function to get more information related to the underlying event. There are various accessor methods which can be called when the cursor is at particular event.

Returning String Representations

Because the next method only returns integers corresponding to underlying event types, you typically need to map these integers to string representations of the events; for example:

```

public final static String getEventTypeString(int eventType) {
    switch (eventType) {
        case XMLEvent.START_ELEMENT:
            return "START_ELEMENT";
        case XMLEvent.END_ELEMENT:
            return "END_ELEMENT";
        case XMLEvent.PROCESSING_INSTRUCTION:
            return "PROCESSING_INSTRUCTION";
        case XMLEvent.CHARACTERS:
            return "CHARACTERS";
        case XMLEvent.COMMENT:
            return "COMMENT";
        case XMLEvent.START_DOCUMENT:
            return "START_DOCUMENT";
        case XMLEvent.END_DOCUMENT:
            return "END_DOCUMENT";
        case XMLEvent.ENTITY_REFERENCE:

```

```
        return "ENTITY_REFERENCE";
    case XMLEvent.ATTRIBUTE:
        return "ATTRIBUTE";
    case XMLEvent.DTD:
        return "DTD";
    case XMLEvent.CDATA:
        return "CDATA";
    case XMLEvent.SPACE:
        return "SPACE";
    }
    return "UNKNOWN_EVENT_TYPE , " + eventType;
}
```

Building and Running the Cursor Example Using NetBeans IDE

Follow these instructions to build and run the Cursor example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to the *tut-install/javaeetutorial5/examples/stax/* directory.
3. Select the cursor folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the cursor project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:

```
-x 1 BookCatalog.xml
```

8. Click OK.
9. Right-click the cursor project and select Run.

Building and Running the Cursor Example Using Ant

To compile and run the Cursor example using Ant, in a terminal window, go to the *tut-install/javaeetutorial5/examples/stax/cursor/* directory and type the following:

```
ant run-cursor
```

Cursor-to-Event Example

Located in the *tut-install/javaeetutorial5/examples/stax/cursor2event/* directory, `CursorApproachEventObject.java` demonstrates how to get information returned by an `XMLEvent` object even when using the cursor API.

The idea here is that the cursor API's `XMLStreamReader` returns integer constants corresponding to particular events, while the event iterator API's `XMLEventReader` returns immutable and persistent event objects. `XMLStreamReader` is more efficient, but `XMLEventReader` is easier to use, because all the information related to a particular event is encapsulated in a returned `XMLEvent` object. However, the disadvantage of event approach is the extra overhead of creating objects for every event, which consumes both time and memory.

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

Instantiating an XMLEventAllocator

The first step is to create a new `XMLInputFactory` and instantiate an `XMLEventAllocator`:

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
System.out.println("FACTORY: " + xmlif);
xmlif.setEventAllocator(new XMLEventAllocatorImpl());
allocator = xmlif.getEventAllocator();
XMLStreamReader xmlr = xmlif.createXMLStreamReader(filename,
    new FileInputStream(filename));
```

Creating an Event Iterator

The next step is to create an event iterator:

```
int eventType = xmlr.getEventType();
while(xmlr.hasNext()){
    eventType = xmlr.next();
    //Get all "Book" elements as XMLEvent object
    if(eventType == XMLStreamConstants.START_ELEMENT &&
        xmlr.getLocalName().equals("Book")){
        //get immutable XMLEvent
        StartElement event = getXMLEvent(xmlr).asStartElement();
        System.out.println("EVENT: " + event.toString());
    }
}
```

Creating the Allocator Method

The final step is to create the `XMLEventAllocator` method:

```
private static XMLEvent getXMLEvent(XMLStreamReader reader)
    throws XMLStreamException {
    return allocator.allocate(reader);
}
```

Building and Running the Cursor-to-Event Example Using NetBeans IDE

Follow these instructions to build and run the Cursor-to-Event example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to the *tut-install/javaeetutorial5/examples/stax/* directory.
3. Select the *cursor2event* folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the *cursor2event* project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:

BookCatalog.xml

8. Click OK.
9. Right-click the *cursor2event* project and select Run.

Note how the Book events are returned as strings.

Building and Running the Cursor-to-Event Example Using Ant

To compile and run the Cursor-to-Event example using Ant, in a terminal window, go to the *tut-install/javaeetutorial5/examples/stax/cursor2event/* directory and type the following:

```
ant run-cursor2event
```

Event Example

Located in the *tut-install/javaeetutorial5/examples/stax/event/* directory, *EventParse.java* demonstrates how to use the StAX event API to read an XML document.

Creating an Input Factory

The first step is to create a new instance of *XMLInputFactory*:

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

Creating an Event Reader

The next step is to create an instance of *XMLEventReader*:

```
XMLEventReader r = factory.createXMLEventReader(filename,
    new FileInputStream(filename));
```

Creating an Event Iterator

The third step is to create an event iterator:

```
XMLEventReader r = factory.createXMLEventReader(filename,
    new FileInputStream(filename));
while(r.hasNext()) {
    XMLEvent e = r.nextEvent();
    System.out.println(e.toString());
}
```

Getting the Event Stream

The final step is to get the underlying event stream:

```
public final static String getEventTypeString(int eventType) {
    switch (eventType) {
        case XMLEvent.START_ELEMENT:
            return "START_ELEMENT";
        case XMLEvent.END_ELEMENT:
            return "END_ELEMENT";
        case XMLEvent.PROCESSING_INSTRUCTION:
            return "PROCESSING_INSTRUCTION";
        case XMLEvent.CHARACTERS:
            return "CHARACTERS";
        case XMLEvent.COMMENT:
            return "COMMENT";
        case XMLEvent.START_DOCUMENT:
            return "START_DOCUMENT";
        case XMLEvent.END_DOCUMENT:
            return "END_DOCUMENT";
        case XMLEvent.ENTITY_REFERENCE:
            return "ENTITY_REFERENCE";
        case XMLEvent.ATTRIBUTE:
            return "ATTRIBUTE";
        case XMLEvent.DTD:
            return "DTD";
        case XMLEvent.CDATA:
            return "CDATA";
        case XMLEvent.SPACE:
            return "SPACE";
    }
    return "UNKNOWN_EVENT_TYPE " + "," + eventType;
}
```

Returning the Output

When you run the Event example, the EventParse class is compiled, and the XML stream is parsed as events and returned to STDOUT. For example, an instance of the Author element is returned as:

```
<['http://www.publishing.org']::Author>  
  Dhirendra Brahmachari  
</['http://www.publishing.org']::Author>
```

Note in this example that the event comprises an opening and closing tag, both of which include the namespace. The content of the element is returned as a string within the tags.

Similarly, an instance of the Cost element is returned as:

```
<['http://www.publishing.org']::Cost currency='INR'>  
  11.50  
</['http://www.publishing.org']::Cost>
```

In this case, the currency attribute and value are returned in the opening tag for the event.

Building and Running the Event Example Using NetBeans IDE

Follow these instructions to build and run the Event example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to the *tut-install/javaeetutorial5/examples/stax/* directory.
3. Select the event folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the event project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:
BookCatalog.xml
8. Click OK.
9. Right-click the event project and select Run.

Building and Running the Event Example Using Ant

To compile and run the Event example using Ant, in a terminal window, go to the *tut-install/javaeetutorial5/examples/stax/event/* directory and type the following:

```
ant run-event
```

Filter Example

Located in the *tut-install/javaeetutorial5/examples/stax/filter/* directory, `MyStreamFilter.java` demonstrates how to use the StAX stream filter API to filter out events not needed by your application. In this example, the parser filters out all events except `StartElement` and `EndElement`.

Implementing the StreamFilter Class

The `MyStreamFilter` class implements `javax.xml.stream.StreamFilter`:

```
public class MyStreamFilter
    implements javax.xml.stream.StreamFilter {
```

Creating an Input Factory

The next step is to create an instance of `XMLInputFactory`. In this case, various properties are also set on the factory:

```
XMLInputFactory xmlif = null ;
try {
    xmlif = XMLInputFactory.newInstance();
    xmlif.setProperty(
        XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES,
        Boolean.TRUE);
    xmlif.setProperty(
        XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES,
        Boolean.FALSE);
    xmlif.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE,
        Boolean.TRUE);
    xmlif.setProperty(XMLInputFactory.IS_COALESCING,
        Boolean.TRUE);
} catch (Exception ex) {
    ex.printStackTrace();
}
System.out.println("FACTORY: " + xmlif);
System.out.println("filename = "+ filename);
```

Creating the Filter

The next step is to instantiate a file input stream and create the stream filter:

```
FileInputStream fis = new FileInputStream(filename);

XMLStreamReader xmlr = xmlif.createFilteredReader(
    xmlif.createXMLStreamReader(fis), new MyStreamFilter());

int eventType = xmlr.getEventType();
printEventType(eventType);
while(xmlr.hasNext()) {
    eventType = xmlr.next();
    printEventType(eventType);
}
```

```
        printName(xmlr,eventType);
        printText(xmlr);
        if (xmlr.isStartElement()) {
            printAttributes(xmlr);
        }
        printPIData(xmlr);
        System.out.println("-----");
    }
}
```

Capturing the Event Stream

The next step is to capture the event stream. This is done in basically the same way as in the Event example.

Filtering the Stream

The final step is to filter the stream:

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

Returning the Output

When you run the Filter example, the `MyStreamFilter` class is compiled, and the XML stream is parsed as events and returned to STDOUT. For example, an `Author` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Author
HAS NO TEXT
HAS NO ATTRIBUTES
-----
EVENT TYPE(2):END_ELEMENT
HAS NAME: Author
HAS NO TEXT
-----
```

Similarly, a `Cost` event is returned as follows:

```
EVENT TYPE(1):START_ELEMENT
HAS NAME: Cost
HAS NO TEXT

HAS ATTRIBUTES:
  ATTRIBUTE-PREFIX:
  ATTRIBUTE-NAME: currency
  ATTRIBUTE-VALUE: USD
  ATTRIBUTE-TYPE: CDATA
```

```

-----
EVENT TYPE(2):END_ELEMENT
HAS NAME: Cost
HAS NO TEXT
-----

```

See “[Iterator API](#)” on page 541 and “[Reading XML Streams](#)” on page 549 for a more detailed discussion of StAX event parsing.

Building and Running the Filter Example Using NetBeans IDE

Follow these instructions to build and run the Filter example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to the *tut-install/javaeetutorial5/examples/stax/* directory.
3. Select the *filter* folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the *filter* project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:


```
-f BookCatalog.xml
```
8. Click OK.
9. Right-click the *filter* project and select Run.

Building and Running the Filter Example Using Ant

To compile and run the Filter example using Ant, in a terminal window, go to the *tut-install/javaeetutorial5/examples/stax/filter/* directory and type the following:

```
ant run-filter
```

Read-and-Write Example

Located in the *tut-install/javaeetutorial5/examples/stax/readnwrite/* directory, *EventProducerConsumer.java* demonstrates how to use a StAX parser simultaneously as both a producer and a consumer.

The StAX *XMLEventWriter* API extends from the *XMLEventConsumer* interface, and is referred to as an *event consumer*. By contrast, *XMLEventReader* is an *event producer*. StAX supports simultaneous reading and writing, such that it is possible to read from one XML stream sequentially and simultaneously write to another stream.

The Read-and-Write example shows how the StAX producer/consumer mechanism can be used to read and write simultaneously. This example also shows how a stream can be modified and how new events can be added dynamically and then written to a different stream.

Creating an Event Producer/Consumer

The first step is to instantiate an event factory and then create an instance of an event producer/consumer:

```
XMLEventFactory m_eventFactory = XMLEventFactory.newInstance();
public EventProducerConsumer() {
}
...
try {
    EventProducerConsumer ms = new EventProducerConsumer();

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

Creating an Iterator

The next step is to create an iterator to parse the stream:

```
while(reader.hasNext()) {
    XMLEvent event = (XMLEvent)reader.next();
    if (event.getEventType() == event.CHARACTERS) {
        writer.add(ms.getNewCharactersEvent(event.asCharacters()));
    } else {
        writer.add(event);
    }
}
writer.flush();
```

Creating a Writer

The final step is to create a stream writer in the form of a new Character event:

```
Characters getNewCharactersEvent(Characters event) {
    if (event.getData().equalsIgnoreCase("Name1")) {
        return m_eventFactory.createCharacters(
            Calendar.getInstance().getTime().toString());
    }
    //else return the same event
    else {
        return event;
    }
}
```

Returning the Output

When you run the Read-and-Write example, the `EventProducerConsumer` class is compiled, and the XML stream is parsed as events and written back to `STDOUT`. The output is the contents of the `BookCatalog.xml` file described in [“Example XML Document” on page 556](#).

Building and Running the Read-and-Write Example Using NetBeans IDE

Follow these instructions to build and run the Read-and-Write example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select `File→Open Project`.
2. In the Open Project dialog, navigate to the `tut-install/javaeetutorial5/examples/stax/` directory.
3. Select the `readnwrite` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the `readnwrite` project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:
`BookCatalog.xml`
8. Click OK.
9. Right-click the `readnwrite` project and select Run.

Building and Running the Read-and-Write Example Using Ant

To compile and run the Read-and-Write example using Ant, in a terminal window, go to the `tut-install/javaeetutorial5/examples/stax/readnwrite/` directory and type the following:

```
ant run-readnwrite
```

Writer Example

Located in the `tut-install/javaeetutorial5/examples/stax/writer/` directory, `CursorWriter.java` demonstrates how to use the StAX cursor API to write an XML stream.

Creating the Output Factory

The first step is to create an instance of `XMLOutputFactory`:

```
XMLOutputFactory xof = XMLOutputFactory.newInstance();
```

Creating a Stream Writer

The next step is to create an instance of `XMLStreamWriter`:

```
XMLStreamWriter xtw = null;
```

Writing the Stream

The final step is to write the XML stream. Note that the stream is flushed and closed after the final `EndDocument` is written:

```
xtw = xof.createXMLStreamWriter(new FileWriter(fileName));
xtw.writeComment("all elements here are explicitly in the HTML namespace");
xtw.writeStartDocument("utf-8", "1.0");
xtw.setPrefix("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "html");
xtw.writeNamespace("html", "http://www.w3.org/TR/REC-html40");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "head");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "title");
xtw.writeCharacters("Frobnostication");
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "body");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "p");
xtw.writeCharacters("Moved to");
xtw.writeStartElement("http://www.w3.org/TR/REC-html40", "a");
xtw.writeAttribute("href", "http://frob.com");
xtw.writeCharacters("here");
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndElement();
xtw.writeEndDocument();
xtw.flush();
xtw.close();
```

Returning the Output

When you run the Writer example, the `CursorWriter` class is compiled, and the XML stream is parsed as events and written to a file named `dist/CursorWriter-Output`:

```
<!--all elements here are explicitly in the HTML namespace-->
<?xml version="1.0" encoding="utf-8"?>
<html:html xmlns:html="http://www.w3.org/TR/REC-html40">
<html:head>
<html:title>Frobnostication</html:title></html:head>
<html:body>
<html:p>Moved to <html:a href="http://frob.com">here</html:a>
</html:p>
</html:body>
</html:html>
```

In the actual `dist/CursorWriter-Output` file, this stream is written without any line breaks; the breaks have been added here to make the listing easier to read. In this example, as with the

object stream in the Event example, the namespace prefix is added to both the opening and closing HTML tags. Adding this prefix is not required by the StAX specification, but it is good practice when the final scope of the output stream is not definitively known.

Building and Running the Writer Example Using NetBeans IDE

Follow these instructions to build and run the Writer example on your Application Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog navigate to the *tut-install/javaeetutorial5/examples/stax/* directory.
3. Select the writer folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the writer project and select Properties. The Project Properties dialog is displayed.
7. Enter the following in the Arguments field:
-f dist/CursorWriter-Output
8. Click OK.
9. Right-click the writer project and select Run.

Building and Running the Writer Example Using Ant

To compile and run the Writer example using Ant, in a terminal window, go to the *tut-install/javaeetutorial5/examples/stax/writer/* directory and type the following:

```
ant run-writer
```

Further Information about StAX

For more information about StAX, see:

- Java Community Process page:
<http://jcp.org/en/jsr/detail?id=173>.
- W3C Recommendation “Extensible Markup Language (XML) 1.0”:
<http://www.w3.org/TR/REC-xml>
- XML Information Set:
<http://www.w3.org/TR/xml-info/>
- W3C Recommendation “Document Object Model”:

<http://www.w3.org/DOM/>

- SAX “Simple API for XML”:

<http://www.saxproject.org/>

- DOM “Document Object Model”:

<http://www.w3.org/>

[TR/2002/WD-DOM-Level-3-Core-20020409/core.html#ID-B63ED1A3](http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/core.html#ID-B63ED1A3)

- W3C Recommendation “Namespaces in XML”:

<http://www.w3.org/TR/REC-xml-names/>

For some useful articles about working with StAX, see:

- Jeff Ryan, “Does StAX Belong in Your XML Toolbox?”:

[http://www.developer.com/](http://www.developer.com/xml/article.php/3397691/Does-StAX-Belong-in-Your-XML-Toolbox.htm)

[xml/article.php/3397691/Does-StAX-Belong-in-Your-XML-Toolbox.htm](http://www.developer.com/xml/article.php/3397691/Does-StAX-Belong-in-Your-XML-Toolbox.htm)

- Elliotte Rusty Harold, “An Introduction to StAX”:

<http://www.xml.com/pub/a/2003/09/17/stax.html>