Streaming API For XML JSR-173 Specification

Java Community Process
http://jcp.org

BEA Systems, Inc.
2315 North First Street
San Jose, CA 95131
U.S.A. 408.570.8000

Final v1.0
October 8th, 2003
Comments to: jsr-173-comments@jcp.org

Streaming API for XML (JSR-173) for Java™ Specification ("Specification")

Version: 1.0

Status: FCS

Release: October, 2003

NOTICE; LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. BEA hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under BEA's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

2. License for the Distribution of Compliant Implementations. BEA also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 3 below, patent rights it may have covering the Specification to create and/or distribute an implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality, and (b) passes the Technology Compatibility Kit for such Specification ("Compliant Implementation").

3. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by BEA and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against BEA that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL BEA OR ITS BEAS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF BEA AND/OR ITS BEAS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend BEA and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your application or applet written to and/or Your implementation of the Specification; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide BEA with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant BEA a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Contents

CHAPTER 1

Introduction

This specification describes the Streaming API for XML (StAX), a bi-directional API for reading and writing XML. This document along with the associated API documentation is the formal specification for JSR-173 [`http://jcp.org/jsr/detail/173.jsp`], which is being worked on under the Java Community Process [`http://jcp.org`].

The structure of this document is as follows:

- Chapter 1, “Introduction to the Streaming API for XML” gives an overview of the API
- Chapter 2, “Use Cases”, outlines the use cases for the API.
- Chapter 3, “Requirements and Recommendations”, discusses the scope and requirements of the specification.
- Chapter 4, “Overview of the Streaming API for XML”, gives an overview of the structure of the API.
- Chapter 5, “Cursor API”, discusses the cursor-style API.
- Chapter 6, “Event Iterator API”, discusses the iterator style API.
- Chapter 7, “Future Work”, outlines future directions for the specification.
- Chapter 8, “References”, contains references to related specifications.

1.1 Benefits of the Streaming API for XML

The Streaming API for XML gives parsing control to the programmer by exposing a simple iterator based API and an underlying stream of events. Methods such as `next()` and `hasNext()` allow an application developer to ask for the next event (pull the event) rather than handling the event in a callback. This gives a developer more procedural control over the processing of the XML document. The Streaming API also allows the programmer to stop processing the document, skip ahead to sections of the document, and get subsections of the document.

Processing XML has become a standard function in most computing environments. Two main approaches exist: 1) the Simple API for XML processing (SAX) and 2) the Document Object Model (DOM). SAX is a low-level parsing API while DOM provides a random-access tree-like structure. This document specifies a new API for parsing and streaming XML between applications in an efficient manner. Efficient XML processing is fundamental for several areas of computing, such as XML based RPC and Data Binding (See Chapter 6, “References”, for related information).

To use SAX, a programmer writes handlers (objects that implement the various SAX handler APIs) that receive callbacks during the processing of an XML document. The main benefits of this style of XML document processing are that it is efficient, flexible, and relatively low level. One drawback to the SAX API is that the programmer must keep track of the current state of the document in the code each time they process an XML document and thus cannot iteratively process it. Another drawback to SAX is that the entire document needs to be parsed at one time.

DOM provides APIs that allow random access and manipulation of an in-memory XML document. At first glance this seems like a win for the application developer. However, this perceived simplicity comes at a very high cost: performance. For very large documents one may be required to read the entire document into memory before taking appropriate actions based on the data.

The Streaming API for XML consists of two styles: A low-level cursor API, designed for creating object models and a higher-level event iterator API, designed to be used in pipelines and be easily extensible.

1.2 Design Goals of the API

The design goals of the Streaming API for XML are as follows:

- Specify symmetrical APIs for reading and writing XML using a streaming paradigm.
- Define an efficient API for reading XML.
- Define APIs that are simple to use for development of Java applications that parse XML.
- Define a clean separation between the reading and writing sides of the API.
- Keep the design of the APIs and mechanisms extensible and modular.
- The API should have a mode that requires as few objects as possible to be created.
- The API should not preclude support for J2ME.

1.3 Expert Group Goals

The goals of the expert group are as follows:

- The API specification, reference implementation, and TCK must be completed in a reasonable amount of time.

  Although reaching GA as quickly as possible is a desirable goal, we must give ourselves enough time to create a quality product.

- Minimize undefined behavior.

  The specification should discuss all aspects of the API, both what is required of an implementation, and what is not required. There should be no ambiguity about particular features of the API. An implementation of the API is compliant with this spec if it implements all required features.

- Make as many features as possible non-optional.

  For standardization reasons, it is clearer and more effective when features are required in an implementation of a specification rather than having lots of optional features. Different implementations of a specification, although internally different, are externally very similar, making them more user-friendly.

Out-of-scope features for the 1.0 version of the Streaming API for XML include:

- Specifying a validation API. Validation will be done in the layer above the streaming parser. This does not preclude passing validation parameters to an underlying parser.
- Specific dependence on an XML grammar.
- Support for applications that transform or edit a DTD.

1.4 Conventions

Within this specification the words should and must are defined as follows:

- *should*

  It is recommended but not required that conforming implementations behave as described.

- *must*

  Conforming implementations are required to behave as described.

All examples in this specification are for illustrative purposes and are non-normative. If an example conflicts with the normative prose the prose always takes precedence. A significant portion of the specification resides in the documentation of the API (javadoc). The javadoc is normative.

1.5 Expert Group Members

- Arnaud Blandin, Intalio, Inc.
- Andy Clark, Apache
- James Clark
- Christopher Fry, BEA Systems [Specification Lead]
- Stefan Haustein
- Simon Horrell, Developmentor
- K Karun, Oracle
- Glenn Marcy, IBM
- Gregory M. Messner, Breeze Factor
- Aleksander Slominski
- David Stephenson, Hewlett-Packard
- James Strachan
- Anil Vijendran, Sun Microsystems

1.6 Acknowledgments

Sam Pullara, Eduardo Pelegri-Llopart, Juliet Shackell, David Bau, Scott Ziegler, Scott Dietzen, Todd Karakashian, Adam Messinger, Beverley Talbot, Elliotte Rusty Harold.

The initial goal for this JSR was to provide a standard for application programmers to stream XML between application boundaries in a fast and elegant manner. Some of the work standardized here arose directly from work done on BEA's XML and WebServices team. The foundation of the RI and many of the ideas found in this work arose directly from Aleksander Slomininski's work on XPP. Many members of the expert group contributed significant effort and time to this JSR, especially Andy Clark, Arnaud Blandin, David Stephenson, Aleksander Slominski, Anil Vijendran, Joe Fialli, K Karun, and Glenn Marcy.

1.7 Status

This document is the Final Version [V1.0] of the Streaming API for XML specification.

CHAPTER 2

Use Cases

This chapter describes the following use cases for the Streaming API for XML:

- Data Binding use cases in Section 2.1 “Data Binding” on page 2-18.
- Web Service use cases in Section 2.2 “SOAP Message Processing” on page 2-20.
- Data source use cases are discussed in Section 2.3 “Virtual Data Source” on page 2-21.
- Writing custom parsers manually in Section 2.4 “Parsing a Specific XML Vocabulary” on page 2-21.
- Pipelining XML procesor use cases in Section 2.5 “Pipelined XML Processing” on page 2-22.

These use cases are presented as general cases that the API should satisfy.

2.1 Data Binding

Data binding is a two-way process that reads and writes XML (unmarshaling and marshaling) to and from a programming language data structure. The following section discusses unmarshaling and marshaling in more detail, as well as other uses for a streaming API in data binding.

2.1.1 Unmarshaling an XML Document

An object model, constructed by the data-binding layer, holds the information contained in an XML document. A data-binding framework can be more efficient if it can access the content of an XML document directly without having to reconstruct the DOM that represents it. By viewing the XML document as a stream of tokens, the data-binding framework can predict the next token, or accept a set of tokens in a given state as it constructs the object tree. SAX provides similar functionalities; however, because SAX fires events, it requires the

data-binding framework to perform state management to keep track of the context in which each of the events are being fired. By employing a pull-based architecture, one is able to simplify the code base and therefore reduce the memory overhead and improve performance.

2.1.2 Marshaling an XML Document

Given a Java object model, the application needs to create the corresponding XML.

A concrete example of this occurs during creation of a SOAP message. Given a Java object model that represents data, an application needs to send a SOAP message that encapsulates the data. One component of the application is responsible for writing the various headers while another component of the application is responsible for writing the RPC body parts. Although the two components are not tightly tied to each other, they should share the same API to write the XML output.

2.1.3 Parallel Document Processing

The Streaming API for XML allows an application to process two documents at the same time. For example: when document X includes or imports document Y, the application can process document Y while processing X. This case is common when the application is reading documents such as XML Schemas or WSDL documents.

2.1.4 Wireless Communication

Small devices may need only a small part of a large XML document. When the device requests a subsection of a document, the document must be unmarshaled and processed.

This process should be fast and optimized. For example: A mobile phone user asks for a quote tag and the server might send a complex XML file that can be used by a broker, a trader or a general user. The mobile phone application should be able to access quickly only the information needed without parsing the entire document. This functionality enables micro- and macro-applications to share the same standard XML documents.

This use case implies the ability to filter specific tokens from the XML stream of events for incremental processing. This ability is needed to process large amounts of data from a record oriented stream, without a large memory requirement.

2.2 SOAP Message Processing

SOAP is an XML message format that can be used as a transport for remote procedure calls (RPC).

2.2.1 Use Case Definition

From a Java server-side perspective, processing an RPC-encoded SOAP message requires the message to be parsed and a reply to be constructed. During the inbound processing the application must:

1. Identify the proper method to invoke.
2. Unmarshal the arguments to the method from XML to Java.
3. Invoke the method.
4. Construct a SOAP response envelope.
5. Marshal the output of the method into the SOAP response envelope.

In some sense the SOAP use case is a superset of the Data Binding use case (Section 2.1 “Data Binding” on page 2-18) because it involves marshaling and unmarshaling XML and Java types.

This use case highlights that the XML Streaming API must be bi-directional to allow parsing and creation of messages. It also points out the need for efficiency since RPC-style invokes should have little or no overhead.

There are two related but not identical scenarios in processing SOAP messages:

- Parsing simple predictable structure that has no forward references (for example an array or linked list).

  This allows for fully streaming approach as SOAP application is consuming events. The stream can be consumed in streaming mode when it can discard parsed XML input.

- Parsing graph representation that has forward links (and possibly circular references)

  The application must record each seen XML event and be able to work correctly with potentially very complex element graph - this can be accomplished by recording events into a buffer or by using a document object model such as DOM or JDOM.

2.2.2 Parsing WSDL

A WSDL document describes the interface of a Web service. A WSDL parser is required to import other WSDL documents and to read XML Schema documents. XML Schema documents can import other schemas. Using an XMLEventReader allows application code to easily include and process other documents concurrently.

2.3 Virtual Data Source

A virtual XML source is data stored as XML, but not necessarily in its serialized format.

2.3.1 Use Case Definition

In many applications, the XML data is stored or accessed in formats other than its serialized format. The consuming application of the data is given an XML view of the data and can move a pointer around to randomly access the data. The Streaming API for XML can be used to present the XML data as events to the application. It can also be used to skip data that is not relevant to the current processing.

2.3.2 Examples of Virtual Data Sources

Data stored in a database can be viewed as XML. The Streaming API for XML can be used to navigate through the result-set of the database query.

Data stored in Java objects created by XML data binding. The Streaming API for XML can be used to access the XML view of the data in the Java objects without serializing it.

The Streaming API for XML can be used to navigate a DOM tree as a stream of events.

2.4 Parsing a Specific XML Vocabulary

Assume that a user is writing an application that interprets a specific, known XML vocabulary. The application needs to parse instances of that XML vocabulary and produce an in-memory data structure. The user wants to have more flexibility than is available from a data-binding framework, perhaps because the user wants to employ internal structures that

are significantly different from the external XML representation. The XML vocabulary is non-trivial and involves recursion. The XML document may have errors in it that need to be reported to the end-user.

A specific example would be an XHTML browser that needs to parse an XHTML document and generate an internal representation that can be used to display the document.

Using DOM is problematic for this use case because it involves having two simultaneous in-memory representations of the XML document. In addition, DOM does not provide line-number and filename information for error reporting. Using SAX is problematic because it requires an inversion of control that prevents writing a natural recursive descent parser. Use of StAX allows the user to write a recursive descent parser to process a specific XML vocabulary without inverting the control flow of the application.

2.5 Pipelined XML Processing

It is often desirable to decompose a complex XML processing application into a pipeline consisting of multiple XML-to-XML transformations. It is inefficient if each stage in a processing pipeline has to parse and serialize XML; it is better for each stage to communicate using an API. The first stage in a pipeline is often an XML parsing stage and the last stage is often an XML serializing stage. Pipeline stages should be composable in a modular way: a particular stage should not rely on where in the pipeline it is being used. Assume that the API has two interfaces: XMLEventReader and XMLEventWriter. We can distinguish three kinds of pipeline stages:

a. A pipeline stage may be sufficiently complex that is is naturally written as something that reads an XMLEventReader and writes an XMLEventWriter; such a pipeline stage would require a separate thread.

   Example: An XSLT Processor.

b. A pipeline stage may be sufficiently simple that it can be written as an implementation of an XMLEventReader interface that reads from an XMLEventReader representing the previous stage.

   Example: A module that strips insignificant whitespace according to some rules (e.g. if two start-tags are separated only by whitespace and there is not an inherited xml:space="preserve" attribute, then remove the whitespace). In this case the stripping module may have to look ahead to the next event in order to determine how to treat the current event. Another example is a module that validates the input according to certain rules (perhaps hard-coded or specified by some sort of schema). In this case the validator may need to report errors to the user.

c. Similar to (b), a pipeline stage may be written as an implementation of an XMLEventWriter that writes to an XMLEventWriter representing the next stage.

   Example: A module that pretty-prints XML by indenting. It might also be useful to have validation implemented in this way.

Implementations of type (b) and (c) will typically be much more efficient than (a), but may be inconvenient for complex processing.

CHAPTER 3

Requirements

This chapter specifies the proposed scope and requirements for the Streaming API for XML. All requirements are listed as initially proposed during the requirements gathering phase of the specification. Several requirements were not met in the first version of the specification (these features have been marked as optional) and may be addressed by a future version of the specification.

R01 Easy and Intuitive to Use

Programmers who are familiar with other Java APIs must be able to easily and quickly learn how to use the XML Streaming API, even if they are not familiar with other XML processing APIs, such as SAX or DOM. The interfaces should be natural and not contain any surprises.

R02 Symmetrical Bi-Directional API

The API can both read and write XML documents using the same representation of the XML.

R03 Roundtrip Namespace Prefixes

When parsing an XML document that uses namespace prefixes, the API must be able to create a new XML document that uses the same namespace prefixes as the original document.

R04 Support Optional Namespace Processing

Backward compatibility for XML 1.0 requires that namespace processing be optional. This requirement is not required for every implementation. Each implementation *must* support namespaces. An implementation may provide a mode which turns off namespace processing, however this is not required.

R05 Maintain Scoped Mapping Between Namespace Prefixes and Corresponding URIs

An application that uses the API does not have to keep track of namespace prefixes, their scope, and the URI to which they they are mapped. The API will have a method to look up this information for a given prefix.

R06 Streaming XML Information Set Properties

The API must support all the properties of the information items within the XML information set of a given XML document that can be processed in a streaming fashion.

R07 Efficient Implementation

The API must allow as efficient an implementation as possible.

R08 Pluggability

The API must define interfaces and use factories to get a particular implementation of the API. This does not imply that a user of the API can use objects created by different implementations of the specification with another implementation.

R09 Support for JAXP

The API must allow support for the `javax.xml.transform.Result` and `javax.xml.transform.Source` interfaces from the *Java API for XML Processing* (JAXP) specification. The API provides support for JAXP interfaces, implementations may choose not to support these methods.

R010 Default Namespace Prefix to URI Mappings

When creating an XML document, the API must provide an option to create default prefix mappings to URIs if none are supplied by the application.

R011 Pre-Parse Configuration

The API must provide the ability to configure the processor to stop a subset of information items from being delivered (for example ProcessingInstructions, Comments & Ignorable Whitespace).

R012 Chaining

The API will provide the ability to plug together stream readers and stream writers.

R013 DTD Support

The API will provide the same level of support for DTDs as does the SAX 2.0 specification.

R014 Support for other XML Specifications

The API must support XML 1.0 and XML Namespaces.

R015 Support Feature Negotiation

The API will provide a way to query and register java.lang.Object properties tied to an arbitrary String.

R016 Entity Resolution

The API must specify an entity resolver interface.

R017 Registration of Non-Fatal Warnings

An application that uses the API will be able to generate non-fatal warnings about the structure or content of an XML document while it is being parsed, without actually stopping the parsing of the document.

R018 Location Information

The API must provide a way to get location information, such as line and column numbers.

R019 Bi-Directional with DOM

An XML stream of events will be able to iterate over a DOM. There will also be a way to create a DOM from an XML stream. This does not imply the ability to roundtrip exactly an XML document represented by the DOM.

This requirement is optional in this version of the specification.

R020 Bi-Directional with SAX

When an application registers a SAX `ContentHandler` with the API, the API invokes the appropriate `ContentHandler` methods during the parse. The API will have an object that implements the `ContentHandler` interface, allowing it to be registered with a SAX parser.

This requirement is optional in this version of the specification.

R021 Skipping Elements

The API should contain methods to skip to the next element when parsing an XML document.

This requirement is optional in this version of the specification.

R022 Stepping Back in the Stream

The API should provide the ability to go back in the stream to previously delivered information items.

This requirement is optional in this version of the specification.

R023 Filtering While Parsing

The API should provide a way to filter a stream while it is being parsed (as opposed to before the parse, which is required).

The API should have the option to create a filtered stream to stop a subset of information items, such as comments or whitespace, from being delivered

This requirement is optional in this version of the specification.

R024 Roundtrip Internal Entity References

When reading, and then writing, an XML document, the same internal entity reference used in the original document will be used in the new document.

This requirement is optional in this version of the specification.

R025 Support External Parsed Entities

This requirement is optional in this version of the specification.

R026 Seek Function

It is a requirement to be able to skip the next element of a stream. Optionally, an implementation can include the ability to skip ahead to a particular element.

This requirement is optional in this version of the specification.

R027 Substreams (XML Fragments)

The API should be able to get a substream of a parent stream without losing the context of the original parse.

While parsing the parent document, the application should be able to get a substream (an in-memory copy of the stream) at a particular point, then continue parsing the parent document from the same point from which it got the substream.

This requirement is optional in this version of the specification.

3.1 Not Supported

The following requirements are out of scope for the first version of the API; an implementation of any of these features is proprietary to the implementer and is not considered part of the Streaming API specification:

- Provide the ability to read from and write to the same stream.
- Allow random-access to the destination.

  The expert group decided that the write API could allow the cursor to be moved to allow random access to destinations.
- Roundtrip character references.
- Preserve whitespace between attributes.
- Preserve attribute quotes.
- Preserve attribute order.
- Provide only the information that SAX provides.

CHAPTER 4

Overview of the Streaming API for XML

This chapter provides an overview of the two main styles of interface discussed in this specification: event iterator and cursor.

The Streaming API for XML has two basic functions: To allow users to read and write XML as efficiently as possible (cursor API) and to be easy to use, event based, easy to extend, and allow easy pipelining (event iterator API). The event iterator API is intended to layer on top of the cursor API.

The cursor API has two interfaces: `XMLStreamReader` and `XMLStreamWriter`. The event iterator API has two main interfaces: `XMLEventReader` and `XMLEventWriter`. All interfaces are discussed in the following sections. Also discussed in this chapter are creation, configuration, reference resolution, and error reporting.

4.1 Events

Both APIs can be thought of as iterating over a set of events. In the cursor API the events may be unrealized; in the event iterator API the events are realized. An XML document is broken down into the following event granularity by both the cursor and event iterator API:

- StartElement
- EndElement
- Attribute
- Namespace
- Characters
- EntityReference
- ProcessingInstruction

- Comment
- StartDocument
- EndDocument
- DTD
- NotationDeclaration
- EntityDeclaration

For details about these event types, see Section 6.3 “Event Types” on page 6-53.

4.2 Cursor API

The cursor API moves a virtual cursor across the underlying XML data. A cursor can be thought of as an interface that moves over the underlying data and allows access to the underlying state through method calls. The cursor model is supported by the `XMLStreamReader` interface. Events exist in both cursor and event iterator API as abstractions describing the XML Infoset. The event iterator API introduces objects representing the events that one can probe for in cursor API.

The following Java API shows the main methods for reading XML in the cursor approach.

```
// Java
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
 }
```

The writing side of the API has methods that correspond to the reading side for “StartElement” and “EndElement” event types.

```
// Java
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
}
```

4.3 Event Iterator API

The event iterator API has an interface that is very easy to implement and use. The `nextEvent()` method returns an object that can be cached or passed on to another component in the chain of processing.

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

The output side of the event iterator API:

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

4.4 Configuration

Configuration is accomplished by setting various properties on the factories. This allows the implementation to choose the best underlying parser for the desired set of user features.

Implementation specific settings can be passed to the underlying implementation using the setProperty() method on the factories. Implementation specific settings can be queried using the getProperty() method on the factories.

4.5 Creating Readers, Writers, and Events

An application programmer can create the various readers, writers and events using the `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory`.

4.5.1 XMLInputFactory

The XMLInputFactory allows the application programmer to configure the implementation instances of processors created by the factory. One creates an instance of the abstract class `XMLInputFactory` by calling the `newInstance()` method on the `XMLInputFactory` class. The static method `XMLInputFactory.newInstance()` creates a new factory instance. This method uses the following ordered lookup procedure to determine the `XMLInputFactory` implementation class to load (for more details see JAXP):

1. Use the `javax.xml.stream.XMLInputFactory` system property.
2. Use the properties file lib/xml.stream.properties in the JRE directory. This configuration file is in standard java.util.Properties format and contains the fully qualified name of the implementation class with the key being the system property defined in step 1.
3. Use the Services API (as detailed in the JAR specification), if available, to determine the classname. The Services API looks for a classname in the file META-INF/services/ javax.xml.stream.XMLInputFactory in jars available to the runtime.
4. Platform default `XMLInputFactory` instance.

Once an application has obtained a reference to a `XMLInputFactory` it can use the factory to configure and obtain stream instances.

4.5.1.1 Supported Properties of XMLInputFactory

javax.xml.stream.isValidating:

Function: Turns on implementation specific validation.

Type: Boolean

Default Value: False

Required: No

javax.xml.stream.isCoalescing:

Function: Requires the processor to coalesce adjacent character data .

Type: Boolean

Default Value: False

Required: Yes

javax.xml.stream.isNamespaceAware:

Function: Turns off namespace support. All implementations must support namespaces supporting non-namespace aware documents is optional.

Type: Boolean

Default Value: True

Required: No

javax.xml.stream.isReplacingEntityReferences:

Function: 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.

Type: Boolean

Default Value: True

Required: Yes

javax.xml.stream.isSupportingExternalEntities:

Function: Requires the processor to resolve external parsed entities.

Type: Boolean

Default Value: Unspecified

Required: Yes

javax.xml.stream.reporter:

Function: sets and gets the implementation of the `XMLReporter`

Type: `javax.xml.stream.XMLReporter`

Default Value: Null

Required: Yes

javax.xml.stream.resolver:

Function: sets and gets the implementation of the `XMLResolver` interface

Type: `javax.xml.stream.XMLResolver`

Default Value: Null

Required: Yes

javax.xml.stream.allocator:

Function: sets/gets the implementation of the `XMLEventAllocator` interface

Type: `javax.xml.stream.util.XMLEventAllocator`

Default Value: Null

Required: Yes

4.5.2 XMLOutputFactory

An instance of the abstract class `XMLOutputFactory` is created by calling the `newInstance()` method on the `XMLOutputFactory` class. The algorithm used to obtain the instance is the same as for `XMLInputFactory` but references the `javax.xml.stream.XMLOutputFactory` system property.

4.5.2.1 Supported Properties of XMLOutputFactory

javax.xml.stream.isRepairingNamespaces:

Function: Creates default prefixes and associates them with Namespace URIs.

Type: Boolean

Default Value: False

Required: Yes

4.5.3 XMLEventFactory

One can create an instance of the abstract class `XMLEventFactory` by calling the `newInstance()` method on the `XMLEventFactory` class. This factory references the property `javax.xml.stream.XMLEventFactory` to instantiate the factory. The `XMLEventFactory` creates instances of the various events used in the Streaming API for XML. The algorithm used to obtain the instance is the same as for `XMLInputFactory` but references the `javax.xml.stream.XMLEventFactory` system property.

4.5.3.1 Supported Properties of XMLEventFactory

There are no default properties supported for this factory.

4.6 Resource Resolution

The `XMLResolver` interface provides a way to set the method that resolves resources during the processing of XML contents. The application sets the interface on the `XMLInputFactory`, which subsequently sets the interface on all processors that the instance of the factory creates.

4.7 Error Reporting and Exception Handling

All fatal errors are reported as `javax.xml.stream.XMLStreamExceptions`. Non-fatal errors and warnings are reported using the `javax.xml.stream.XMLReporter` interface.

4.8 Attributes and Namespaces

Attributes are reported using lookup methods and strings in the cursor interface and Attribute and Namespace events in the event iterator interface. A namespace is an attribute; however namespaces are reported separately from attributes in both styles of API. This does not preclude an application that uses this specification from reporting attributes or namespaces as top level events. For example an XPath implementation may report the result of a query for an attribute as a standalone event not associated with a start element.

4.8.1 Optional Namespace Processing

Full support for XML 1.0 requires that namespace processing be optional. All implementations of this specification must support XML 1.0 and the XML Namespace specifications. Implementations may support an option to turn off namespace processing. This means that all names will be reported with the full text of the name in the local name field of the qualified name. Any method that accesses namespace information will return null.

4.8.2 Namespace Bindings

The current binding of the default namespace can be found by looking up the empty string "" from the javax.xml.namespace.NamespaceContext interface. If the default namespace is not bound, the lookup returns null. If it is bound, the lookup returns the current value of the namespace URI.

As per the XML 1.0 specification, the prefix "xml" is bound to "http://www.w3.org/XML/1998/namespace". See the XML Information Set specification.

The "xmlns" prefix is resolved to "http://www.w3.org/2000/xmlns/". See the Namespaces in XML specification.

The namespace context of the current state is available by calling `XMLStreamReader.getNamespaceContext()` or `StartElement.getNamespaceContext()`. These methods return an instance of the `javax.xml.namespace.NamespaceContext` interface.

4.9 XML 1.0 Conformance

An XML processor that implements the Streaming API for XML must be XML 1.0 compliant. At a minimum the processor must follow the XML 1.0 specification in regards to non-validating XML processors. The following list discusses the major areas that a conformant XML processor implementation must consider:

- An XML processor must always provide all characters in a document that are not part of markup to the application.
- A validating XML processor must inform the application which non-markup characters are whitespace appearing within element content.

  The API provides methods to query for whitespace: public boolean XMLStreamReader.isWhiteSpace() and a Space event to report whitespace.

- An XML processor must pass the single character &#A; in place of &#D; or &#D;&#A; appearing in its input.

  This is the job of the processor implementation and must be enforced. When an XMLStream of events is pipelined a processor is not required to renormalize end of line characters.
- An XML processor must normalize the value of attributes.

  This is the job of the underlying processor implementation and must be enforced.
- An XML processor must pass the identifiers of declared unparsed entities and their associated identifiers to the application.

  Unparsed entites are reported on the DTD event and can be queried using the property API on the XMLStreamReader interface.

- An XML processor must report well-formedness errors in the document entity and in any other entities that it reads.

  All well-formedness errors are reported as XMLStreamExceptions. Well-formedness is enforced by the implementation.
- A validating XML processor must report violations of the constraints expressed in the DTD, and failures to fulfill validity constraints. All entities included directly or indirectly by the document entity must be examined.

  Validation errors are reported through the XMLReporter interface.

- An XML processor must pass processing instructions to the application.

  Processing instructions are reported as events.
- An XML processor (necessarily a non-validating one) that does not include the replacement text of an external parsed entity in place of an entity reference must notify the application that it recognized but did not read the entity.

  Unread entities should be reported as entity reference events.

- A validating XML processor must include the replacement text of an entity in place of an entity reference.

  The default behavior of an implementation must be to include the replacement text of an entity. Non-validating conforming implementations should be configurable to allow simple reporting of the reference.

- A validating XML processor must supply the default value of attributes declared in the DTD for a given element type but not appearing in the element's start tag.

  Underlying processor implementations must support attribute defaulting.

4.10 Well-Formedness

All streams that are created from XML `java.io.InputStream` or `java.io.Reader` are required to be well-formed. It is a fatal error if the underlying XML is not-well formed. Applications can take the original well-formed stream and create a malformed stream. For example, given an XML document, an application can create a filter that reports only the StartElements in that document by taking the original well-formed stream and applying a filter to it. The behavior (event structure) of a non-well formed stream created by an application is unspecified.

4.11 Validation

Handling of validation and reporting of validation errors is an implementation defined procedure. All implementations must supply an implementation of an XML 1.0 non-validating parser that supports the Namespaces in XML 1.0 specification.

4.12 J2ME Subset

This section is non-normative and serves as an indication of which classes would be sufficient for a J2ME implementation. The Expert Group has determined that the following subset of the API is what would be suitable for J2ME. How an instance is created under J2ME is currently undefined.

The following classes are sufficient to be the J2ME subset of the API.

`javax.xml.stream.XMLStreamReader`

`javax.xml.stream.XMLStreamWriter`

`javax.xml.stream.XMLStreamException`

`javax.xml.stream.Location`

`javax.xml.stream.XMLStreamConstants`

Other referenced javax interfaces/classes:

`javax.xml.namespace.QName`

`javax.xml.namespace.NamespaceContext`

`javax.xml.XMLConstants`

CHAPTER 5

Cursor API

The cursor API moves a virtual cursor across the XML input and provides accessor methods to the contents pointed to by the virtual cursor. The cursor API is designed to be an efficient and low-level means for constructing object models and representations of the XML where the form of the XML is no longer needed. This chapter discusses the `XMLStreamReader` and `XMLStreamWriter` APIs.

5.1 XMLStreamReader

The `XMLStreamReader` interface provides forward, read-only access to the underlying data structure. A processor that implements the `XMLStreamReader` interface can read a stream or a document. The `XMLStreamReader` interface defines methods that enable one to pull data from XML or skip unwanted events.

`XMLStreamReader` has methods to:

- 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).

At any moment in time an `XMLStreamReader` instance has a current event that the methods of the interface access. Instances are created in the START_DOCUMENT state and this is set to the current event type. A call to `next()` will load the next event.

5.1.1 Reading XML

Use the `XMLStreamReader.next()` method to read the next content event of an XML document. This loads the properties of the next event of the input into the underlying processor. The properties can then be accessed with the `XMLStreamReader.getLocalName()` and `XMLStreamReader.getText()` methods.

An `XMLStreamReader` implementation should include a useful `toString()` implementation that returns a String representation of the current state.

5.1.2 Reading Attributes and Namespaces

When the cursor moves over a `StartElement` event, it reads the name, and associated namespace and attribute definitions. All the attributes declared on the `StartElement` event are available using an index value. The value of an attribute is also available using a direct lookup by namespace URI and local name. The following methods allow the user to get information about attributes and namespaces:

```
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 be accessed using the following methods:

```
int getNamespaceCount();

String getNamespacePrefix(int index);

String getNamespaceURI(int index);
```

Only the namespaces declared on the current `StartElement` are available. The list does not contain previously declared namespaces and does not remove redeclared namespaces.

5.1.2.1 Attribute Value Normalization

The attribute values returned are required to be normalized as per the XML 1.0 specification.

5.1.3 Reading Entity References

Depending on the configuration of the processor, the `XMLStreamReader` either replaces entity references with their text value or returns the entity reference itself.

5.2 XMLStreamWriter

`XMLStreamWriter` defines an interface for writing XML. The `XMLStreamWriter` provides a means for generating XML streams to help build XML documents.

The `XMLStreamWriter` interface has methods to:

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

`XMLStreamWriter` implementations may (but are not required to) check for:

- Invalid element and attribute name characters.
- Unicode characters that do not fit the specified encoding.
- Duplicate attributes.

An implementation is not required to perform well-formedness or validity checks on its input. Implementations may implement strict error checking. An XMLStreamWriter may allow documents with more than one root element to be written. An implementation should clearly document what level of well-formedness and validity checks it supports through the property mechanism provided on the `XMLOutputFactory`.

5.2.1 Escaping Characters

Use the `writeCharacters(...)` method to escape characters such as '&', '<', '>', and ' " 'found in the string and write these to the output mechanism.

5.2.2 Binding Prefixes

The following three options exist for writing prefixes:

- Pass in the actual value to write for the prefix for an event.
- Use the `setPrefix()` method to set the prefix that should be bound to a specific namespace URI.
- Set the property which enables defaulting of declarations for namespace URIs which are not bound to a specific prefix or the default namespace.

If the implementation is not defaulting prefixes, writing an undeclared namespace URI is a fatal error. If the implementation is defaulting prefixes when the implementation encounters an undeclared namespace URI it will associate an implementation defined namespace prefix with the URI. The resulting prefix is written for each instance of the URI encountered.

5.3 Examples of Basic Functionality

These examples are illustrative and non-normative. They are designed to give a basic feel of how to create a factory and get a processor.

5.3.1 XMLStreamReader

This example illustrates how to instantiate an input factory, create a reader and iterate over the elements of an XML document.

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
while(r.hasNext()) {
  r.next();
}
```

5.3.2 XMLStreamWriter

Ths example illustrates 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("foo bar foo");
writer.writeEndElement();
writer.flush();
```

This will result in 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"/>foo bar foo</a>
```

5.4 Summary

The cursor API efficiently reads and writes XML data. Chapter 6 discusses the event iterator API.

CHAPTER 6

Event Iterator API

The Event Iterator API is easy to use and extend. It surfaces a set of events that the application programmer can use to manipulate XML. The events are allocated and no restrictions are placed on their reuse. The events are designed to be easy to filter, buffer, persist and compare. As in the `XMLStreamReader` and `XMLStreamWriter` case, there are a pair of interfaces that read and write XML: `XMLEventReader` and `XMLEventWriter`.

6.1 XMLEventReader

The `XMLEventReader` interface has methods for iterating over XML content: `next()`, `nextEvent()`, `hasNext()` and `peek()` as illustrated in the following code:

```
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;
 ...
}
```

The `next()` method returns the next event on the stream of events (use `nextEvent()` to return a typed `XMLEvent`); the `peek()` method returns this event but does not take it off of the stream. The method `hasNext()` returns true if there are more events to process on the stream. See Section 6.3 “Event Types” on page 6-53 for a discussion of event types.

For example, the following code snippet reads all the events on a stream and prints them.

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

6.1.1 Reading Attributes

Attributes are accessed from their associated `javax.xml.stream.StartElement`:

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

Use the `getAttributes()` method of the `StartElement` interface to get an `Iterator` over all the attributes declared on that `StartElement`.

The `Attribute` interface inherits from `XMLEvent`. This allows implementations (such as XPath) to return attributes as information items.

`xml:space` and `xml:lang` are treated as normal attributes.

6.1.2 Reading Namespaces

Namespaces are read using an `Iterator`. Use the `getNamespaces()` method of the `StartElement` interface to get an `Iterator` over all the namespaces declared on that `StartElement`. This method returns only the namespaces declared on the current element. It does not return any of the attributes declared on the element. The `Namespace` interface inherits from `Attribute`, indicating that a namespace is an `Attribute`, and can be treated as such in an application.

An application can get a copy of the current namespace context by calling the `StartElement.getNamespaceContext()` method. This method returns an instance of the `javax.xml.namespace.NamespaceContext` interface which allows lookup of the current namespaces in scope and the default namespace (if any).

6.1.3 Entity References

An entity reference can be handled in one of two ways. It is strongly recommended that all entity and character references be resolved and reported to the application transparently.

- If `javax.xml.stream.isReplacingEntityReferences` is set to true all entity references are resolved and reported as `Characters` or their replacement value transparently.
- If `javax.xml.stream.isReplacingEntityReferences` is set to false entity references will be reported as an `EntityReference` event.

6.2 XMLEventWriter

The main API for output using the event iterator style interface is as follows:

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

An instance of this interface is created by an `XMLOutputFactory`. Events are then added iteratively to the output stream. An event may not be modified after it has been added to an event writer.

6.2.1 Attribute Handling

Attributes or namespaces can be added to the output stream of events at the top level. The underlying implementation is required to buffer the last `StartElement` until an event other than an attribute or namespace is added. When an attribute or namespace is added it is appended to the buffered `StartElement` event. A call to flush will cause the underlying `XMLEventWriter` to flush all buffered events. There is asymmetry in attribute handling but this makes generating XML much more convenient for application programmers.

6.2.2 Escaping Characters

The method that writes `Characters` events escapes characters such as ‘&’, ‘<‘, ‘>’, and ‘ “ ’ found in the string.

6.2.3 Binding Prefixes

The `XMLEventWriter` implementation adds namespace bindings to its internal namespace map. The `setPrefix( ... )` method can be used to bind a prefix explicitly for use during output. Prefixes are bound in a scoped manner (prefixes bound inside an element go out of scope after the element's corresponding `EndElement` event).

The `getPrefix( ...)` method can be used to discover the current prefix bound to a URI.

6.3 Event Types

The iterator API returns the following event types.

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

Event implementations should include useful implementations of `toString()` suitable for debugging event streams.

For example, the following XML:

```
<!xml version="1.0" encoding="utf-8"!><Hello>world</Hello>
```

would create the following events:

```
EVENT:START_DOCUMENT [<?xml version="1.0" encoding="utf-8"
standalone="yes"?>]
EVENT:START_ELEMENT [<Hello>]
EVENT:CHARACTERS [world]
EVENT:END_ELEMENT [</Hello>]
EVENT:END_DOCUMENT []
```

Implementations of the Streaming API for XML provide ways to filter unwanted events.

6.3.1 Start Document

The `StartDocument` event reports the beginning of a set of XML events and reports the encoding, XML version, and standalone properties.

6.3.2 StartElement

The `StartElement` event encapsulates information about the start of an element. It reports any attributes and namespace declarations declared in the start-tag of the actual XML. It also gives access to the prefix, namespace URI and local name of the start-tag.

6.3.3 EndElement

The `EndElement` event corresponds directly to the end-tag of an element. Namespaces that have gone out of scope are retrievable from this event if those Namespaces have explicitly been set on the associated `StartElement`.

6.3.4 Characters

The `Characters` event corresponds to the following underlying XML entities: CData sections and CharacterData. Ignorable whitespace and significant whitespace (see the W3C Recommendation: XML) are reported as `Characters` events. How entity references are resolved should be configured before instantiating an `XMLEventReader`.

Methods on the `Characters` event allow one to query whether the `Characters` event consists of all whitespace (as defined in the XML 1.0 spec) and whether the `Characters` event is ignorable whitespace.

Whitespace inside element content is reported as a `Characters` event.

6.3.5 Entity Reference

An entity reference can be reported as its own event that allows the application programmer to resolve the entity or pass it through unresolved. The default behavior is to resolve entity references. `EntityReferences` may be reported as independent events or the replacement text can be substituted in and reported as `Characters`.

6.3.6 Processing Instruction

The `ProcessingInstruction` event reports both the target and the data of the underlying processing instruction.

6.3.7 Comment

The text of a comment is reported in a `Comment` event.

6.3.8 End Document

The `EndDocument` event signals the end of the set of XML events.

6.3.9 DTD

The `DTD` event encapsulates information about the Document Type Definition. Validation is implementation specific. The `DTD` event allows the programmer to access the Document Type Definition as a `java.lang.String` and provides a method that allows implementations to return custom objects that encapsulate the information found in the DTD.

6.3.10 Attribute

Some XML processing situations (for example XQuery or XPath query results) require that an attribute be reported as a standalone information item. In general attributes are reported as part of a `StartElement` event. However, an `Attribute` event can occur as a top level element in certain situations (such as the result of an XPath expression). A standalone attribute event will never be encountered as the result of processing a standalone XML document.

6.3.11 Namespace

Namespace declarations can also exist outside of a StartElement and may be reported as a standalone information item. In general Namespaces are reported as part of a StartElement event. When namespaces are the result of an XQuery or XPath expression they may be reported as standalone events.

6.4 Event Reading Example

This example is illustrative and non-normative. It is designed to give a basic feel of how to create a factory and get a processor.

6.4.1 Reading Events

The following example shows how to iterate over the set of events created by parsing an XML document.

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLEventReader r = f.createXMLEventReader(...);
while(r.hasNext()) {
      XMLEvent e = r.nextEvent();
    }
  }
```

6.5 Summary

The `XMLEventReader` and `XMLEventWriter` are designed to be easy to use, extensible, and flexible.

CHAPTER 7

Future Work

As this API gains acceptance it will require updates to this specification.

7.1 Validation

Modules that validate XML Schemas and DTDs would be convenient utilites.

7.2 Virtual Data Sources

The API could be used as a layer over virtual data sources.

CHAPTER 8

References

1. W3C Recommendation “Extensible Markup Language (XML) 1.0”: http://www.w3.org/TR/REC-xml
2. XML Information Set: http://www.w3.org/TR/xml-infoset/
3. JAXB specification: http://java.sun.com/xml/jaxb
4. JAX-RPC specification: http//java.sun.com/xml/jaxrpc
5. W3C Recommendation “Document Object Model”: http://www.w3.org/DOM/
6. SAX “Simple API for XML”: http://www.saxproject.org/
7. DOM “Document Object Model”: http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020409/core.html#ID-B63ED1A3
8. W3C Recommendation “Namespaces in XML”: “http://www.w3.org/TR/REC-xml-names/”