Pro XML Development with Java™ Technology

Ajay Vohra and Deepak Vohra

Apress®

Pro XML Development with Java™ Technology

ISBN-13 (pbk): 978-1-59059-706-4

ISBN-10 (pbk): 1-59059-706-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Chris Mills
Technical Reviewer: Bharath Gowda
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Kim Burton
Indexer: Carol Burbo
Artist: Susan Glinert Stevens
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 2

Parsing XML Documents

An XML document contains structured textual information. We covered the syntactic rules that define the structure of a well-formed XML document in the primer on XML 1.0 in Chapter 1. This chapter is about parsing the structure of a document to extract the content information contained in the document.

We'll start by discussing various objectives for parsing an XML document and by covering various parsing approaches compatible with these objectives. We'll discuss the advantages and disadvantages of each approach and the appropriateness of them for particular applications. We'll then discuss specific parsing APIs that implement these approaches and are defined within JAXP 1.3, which is included in J2SE 5.0, and Streaming API for XML (StAX), which is included in J2SE 6.0. We'll explain each API through code examples. Finally, we'll offer instructions on how to build and execute these code examples within the Eclipse IDE.

Objectives of Parsing XML

Parsing is the most fundamental aspect of processing an XML document. When an application parses an XML document, typically it has three distinct objectives:

- To ensure that the document is well-formed
- To check that the document conforms to the structure specified by a DTD or an XML Schema
- To access, and maybe modify, various elements and attributes specified in the document, in a manner that meets the specific needs of an application

All applications share the first objective. The second objective is not as pervasive as the first but is still fairly standard. The third objective, not surprisingly, varies from application to application. Prompted by the diverse access requirements of various applications, different parsing approaches have evolved to satisfy these requirements. To date, you can take one of three distinct approaches to parsing XML documents:

- DOM[1] parsing
- Push parsing
- Pull parsing

In the next section, we will give an overview of these three approaches and then offer a comparative analysis of them.

[1] You can find the Document Object Model (DOM) Level 3 Core specification at http://www.w3.org/TR/DOM-Level-3-Core/.

Overview of Parsing Approaches

In the following sections, we will give you an overview of the three major parsing approaches from a conceptual standpoint. In later sections, we will discuss specific Java APIs that implement these approaches. We will start with the DOM approach.

DOM Approach

The Document Object Model (DOM) Level 3 Core specification specifies platform- and language-neutral interfaces for accessing and manipulating content and specifies the structure of a generalized document. The DOM represents a document as a tree of Node objects. Some of these Node objects have child node objects; others are leaf objects with no children.

To represent the structure of an XML document, the generic Node type is specialized to other Node types, and each specialized node type specifies a set of allowable child Node types. Table 2-1 explains the specialized DOM Node types for representing an XML document, along with their allowable child Node types.

Table 2-1. *Specialized DOM Node Types for an XML Document*

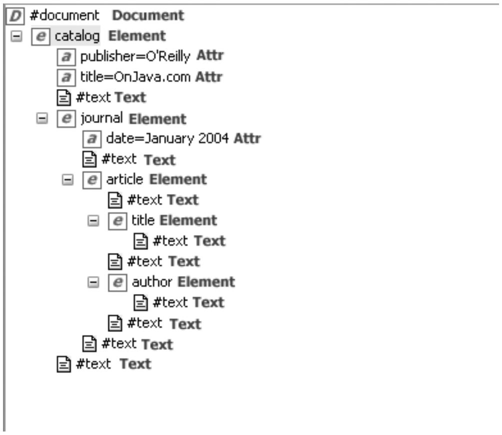
Specialized Node Type	Description	Allowable Child Node Types
Document	Represents an XML document	DocumentType, ProcessingInstruction, Comment, Element(maximum of 1)
DocumentFragment	Represents part of an XML document	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	Represents a DTD for a document	No children
EntityReference	Represents an entity reference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Represents an element	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Attr	Represents an attribute	Text, EntityReference
ProcessingInstruction	Represents a processing instruction	No children
Comment	Represents a comment	No children
Text	Represents text, including whitespace	No children
CDATASection	Represents a CDATA section	No children
Entity	Represents an entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	Represents a notation	No children

The Document specialized node type is somewhat unique in that at most only one instance of this type may exist within an XML document. It is also worth noting that the Document node type is a specialized Element node type and is used to represent the root element of an XML document. Text node types, in addition to representing text, are also used to represent whitespace in an XML document.

Under the DOM approach, an XML document is parsed into a random-access tree structure in which all the elements and attributes from the document are represented as distinct nodes, with each node instantiated as an instance of a specialized node type. So, for example, under the DOM approach, the example XML document shown in Listing 2-1 would be parsed into the tree structure (annotated with specialized node types) shown in Figure 2-1.

Listing 2-1. *Example XML Document*

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog title="OnJava.com" publisher="O'Reilly">
<journal date="January 2004">
   <article>
    <title>Data Binding with XMLBeans</title>
    <author>Daniel Steinberg</author>
   </article>
 </journal>
</catalog>
```



```
D #document Document
⊟ e catalog Element
      a publisher=O'Reilly Attr
      a title=OnJava.com Attr
      #text Text
  ⊟ e journal Element
        a date=January 2004 Attr
        #text Text
    ⊟ e article Element
          #text Text
      ⊟ e title Element
            #text Text
          #text Text
      ⊟ e author Element
            #text Text
          #text Text
        #text Text
      #text Text
```

Figure 2-1. *Annotated DOM tree for example XML document*

The DOM approach has the following notable aspects:

- An in-memory DOM tree representation of the complete document is constructed before the document structure and content can be accessed or manipulated.
- Document nodes can be accessed randomly and do not have to be accessed strictly in document order.
- Random access to any tree node is fast and flexible, but parsing the complete document before accessing any node can reduce parsing efficiency.

- For large documents ranging from hundreds of megabytes to gigabytes in size, the in-memory DOM tree structure can exhaust all available memory, making it impossible to parse such large documents under the DOM approach.
- If an XML document needs to be navigated randomly or if the document content and structure needs to be manipulated, the DOM parsing approach is the most practical approach. This is because no other approach offers an in-memory representation of a document, and although such representation can certainly be created by the parsing application, doing so would be essentially replicating the DOM approach.
- An API for using the DOM parsing approach is available in JAXP 1.3.

Push Approach

Under the push parsing approach, a push parser generates synchronous events as a document is parsed, and these events can be processed by an application using a callback handler model. An API for the push approach is available as SAX[2] 2.0, which is also included in JAXP 1.3. SAX is a read-only API. The SAX API is recommended if no modification or random-access navigation of an XML document is required.

The SAX 2.0 API defines a `ContentHandler` interface, which may be implemented by an application to define a callback handler for processing synchronous parsing events generated by a SAX parser. The `ContentHandler` event methods have fairly intuitive semantics, as listed in Table 2-2.

Table 2-2. *SAX 2.0 `ContentHandler` Event Methods*

Method	Notification
`startDocument`	Start of a document
`startElement`	Start of an element
`characters`	Character data
`endElement`	End of an element
`endDocument`	End of a document
`startPrefixMapping`	Start of namespace prefix mapping
`endPrefixMapping`	End of namespace prefix mapping
`skippedEntity`	Skipped entity
`ignorableWhitespace`	Ignorable whitespace
`processingInstruction`	Processing instruction

[2] You can find information about Simple API for XML at `http://www.saxproject.org/`.

In addition to the ContentHandler interface, SAX 2.0 defines an ErrorHandler interface, which may be implemented by an application to receive notifications about errors. Table 2-3 lists the ErrorHandler notification methods.

Table 2-3. *SAX 2.0 ErrorHandler Notification Methods*

Method	Notification
fatalError	Violation of XML 1.0 well-formed constraint
error	Violation of validity constraint
warning	Non-XML-related warning

An application should make no assumption about whether the endDocument method of the ContentHandler interface will be called after the fatalError method in the ErrorHandler interface has been called.

Pull Approach

Under the pull approach, events are pulled from an XML document under the control of the application using the parser. StAX is similar to the SAX API in that both offer event-based APIs. However, StAX differs from the SAX API in the following respects:

- Unlike in the SAX API, in the StAX API, it is the application rather than the parser that controls the delivery of the parsing events. StAX offers two event-based APIs: a cursor-based API and an iterator-based API, both of which are under the application's control.
- The cursor API allows a walk-through of the document in document order and provides the lowest level of access to all the structural and content information within the document.
- The iterator API is similar to the cursor API but instead of providing low-level access, it provides access to the structural and content information in the form of event objects.
- Unlike the SAX API, the StAX API can be used both for reading and for writing XML documents.

Cursor API

Key points about the StAX cursor API are as follows:

- The XMLStreamReader interface is the main interface for parsing an XML document. You can use this interface to scan an XML document's structure and contents using the next() and hasNext() methods.
- The next() method returns an integer token for the next parse event.
- Depending on the next event type, you can call specific allowed methods on the XMLStreamReader interface. Table 2-4 lists various event types and the corresponding allowed methods.

Table 2-4. *StAX Cursor API Event Types and Allowed Methods*

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

Iterator API

Key points about the StAX iterator API are as follows:

- The XMLEventReader interface is the main interface for parsing an XML document. You can use this interface to iterate over an XML document's structure and contents using the nextEvent() and hasNext() methods.
- The nextEvent() method returns an XMLEvent object.
- The XMLEvent interface provides utility methods for determining the next event type and for processing it appropriately.

The StAX API is recommended for data-binding applications, specifically for the marshaling and unmarshaling of an XML document during the bidirectional XML-to-Java mapping process. A StAX API implementation is included in J2SE 6.0.

Comparing the Parsing Approaches

Each of the three approaches discussed offers advantages and disadvantages and is appropriate for particular types of applications. Table 2-5 compares the three parsing approaches.

Table 2-5. *DOM, SAX, and StAX Comparison*

Parsing Approach	Advantages	Disadvantages	Suitable Application
DOM	Ease of use, navigation, random access, and XPath support	Must parse entire document, memory intensive	Applications that modify structure and content of an XML document, such as visual XML editors*
SAX	Low memory consumption, efficient	No navigation, no random access, no modification	Read-only XML applications, such as document validation
StAX	Ease of use, low memory consumption, application regulates parsing, filtering	No random access, no modification	Data binding, SOAP message processing

* We've written such an editor, which is available at `http://www.nubean.com`.

Before you see some code examples of the three parsing APIs, we'll show how to create and configure an appropriate Eclipse project.

Setting Up an Eclipse Project

In the following sections, we will show how to set up an Eclipse project and populate it with the contents needed to build and execute code examples related to the three parsing approaches discussed in this chapter. Even though in later sections we will discuss each parsing approach separately, here we will show how to prepare the Eclipse project for all three parsing approaches at once.

Example XML Document

To take any of the parsing approaches, the first element you need is an XML document. To that end, you can use the example XML document shown in Listing 2-2.

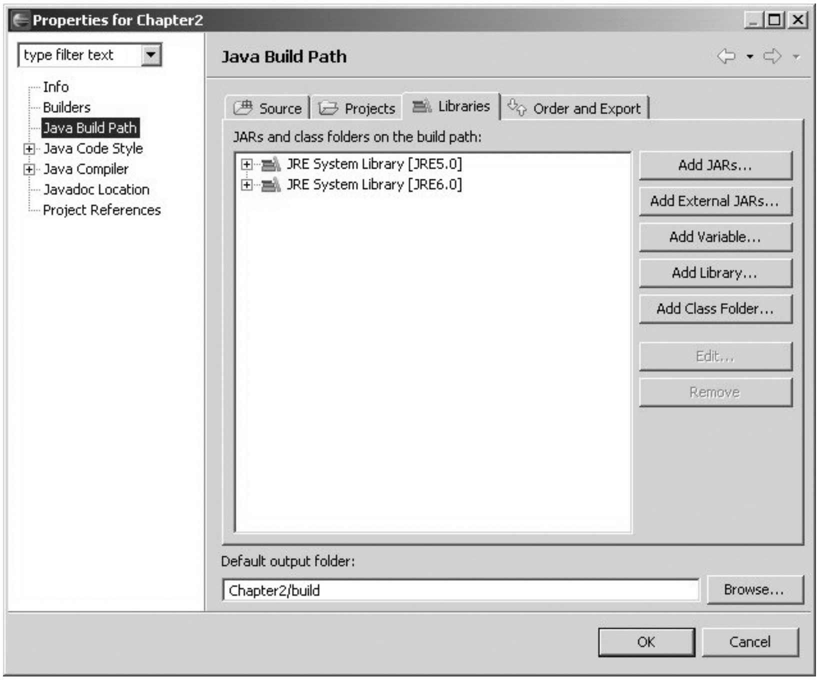
Listing 2-2. *catalog.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog title="OnJava.com" publisher="O'Reilly">
<journal date="January 2004">
   <article>
    <title>Data Binding with XMLBeans</title>
    <author>Daniel Steinberg</author>
   </article>
 </journal>
 <journal date="Sept 2005">
   <article>
    <title>What Is Hibernate</title>
    <author>James Elliott</author>
   </article>
 </journal>
</catalog>
```

J2SE, Packages, and Classes

To build and execute these examples, you need to make sure you have the J2SE 5.0 software development kit (SDK)[3] and the J2SE 6.0 SDK (code-named Mustang[4]) installed on your machine.

Next, download the Chapter2 project from the Apress website (http://www.apress.com) and import it, as explained in detail in Chapter 1. Importing the project is the quickest way to run the example applications, because all the packages and files in the project get created automatically and the Java build path gets set automatically. Please verify that the Java build path is as shown in Figure 2-2 and the overall project structure is as shown in Figure 2-3.



Properties for Chapter2

Java Build Path

JARs and class folders on the build path:

JRE System Library [JRE5.0]
JRE System Library [JRE6.0]

Default output folder: Chapter2/build

Figure 2-2. *Chapter2 project Java runtime environments (JREs)*

[3] You can download the J2SE 5.0 SDK from http://java.sun.com/j2se/1.5.0/download.jsp.

[4] You can download the snapshot release of Mustang from https://mustang.dev.java.net/.

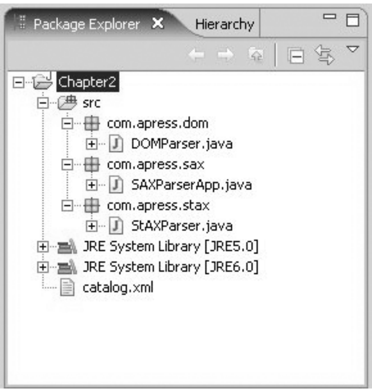


Figure 2-3. *`Chapter2` project directory structure*

Parsing with the DOM Level 3 API

The DOM Level 3 API, which is part of the JAXP 1.3 API, represents an XML document as a tree of DOM nodes. Each node in this tree is a specialized `Node` object that is an instance of one of the specialized `Node` types listed in Table 2-1. The following packages and classes are essential parts of any application that uses the DOM Level 3 API:

- The classes and interfaces representing the DOM structure of an XML document are in the `org.w3c.dom` package, which must be imported by an application using the DOM API.
- The `NodeList` interface represents an ordered list of nodes. A `NamedNodeMap` represents an unordered set of nodes, such as attributes of an element. Both these classes are useful in traversing the DOM tree representing an XML document.
- The XML document–parsing API is in the `javax.xml.parsers` package. This is an essential package and must be imported by an application parsing an XML document using the DOM API.
- An application needs to import the `org.xml.sax` package so it can access the `SAXException` and `SAXParseException` classes, which are used in error handling. This reference to the SAX API within the DOM API may seem out of place. However, this reliance of the DOM API on the SAX API is specified by JAXP 1.3 and is basically an attempt to reuse the SAX API where appropriate.

DOM API Parsing Steps

To parse an XML document using the DOM API, you need to follow these steps:

1. Create a DOM parser factory.
2. Use the parser factory to instantiate a DOM parser.
3. Use the DOM parser to parse an XML document and create a DOM tree.
4. Access and manipulate the XML structure and content by accessing the DOM tree.

The DocumentBuilder class implements the DOM parser. The steps to instantiate a DocumentBuilder object are as follows:

1. Create a DocumentBuilderFactory object using the static method newInstance(). The DocumentBuilderFactory class is a factory API for generating DocumentBuilder objects.
2. Create a DocumentBuilder object by invoking the newDocumentBuilder() static method on the DocumentBuilderFactory object.

The DocumentBuilder parser creates an in-memory DOM structure from an XML document. If you want to handle validation errors during parsing, you need to define a class that implements the ErrorHandler interface shown in Table 2-3 and set an instance of this error handler class on the parser. Listing 2-3 shows an example class that implements the ErrorHandler interface.

Listing 2-3. *Implementing ErrorHandler*

```
class ErrorHandlerImpl implements org.xml.sax.ErrorHandler {
  public void error(SAXParseException exception)
               throws SAXException{
     // application-specific logic
   }

   public void fatalError(SAXParseException exception)
                 throws SAXException{
     // application-specific logic
    }

    public void warning(SAXParseException exception)
                 throws SAXException{
       // application-specific logic
     }
}
```

Listing 2-4 shows the complete code sequence for creating a DOM parser object that will validate a document and use an instance of the ErrorHandlerImpl class for error handling.

Listing 2-4. *Complete Code Sequence to Instantiate the Factory*

```
//Create a DocumentBuilderFactory
DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
//Create a DocumentBuilder
DocumentBuilder documentBuilder=factory.newDocumentBuilder();
//Create and set an ErrorHandler
ErrorHandlerImpl errorHandler=new ErrorHandlerImpl();
documentBuilder.setErrorHandler(errorHandler);
```

A parser can parse an XML document from a File, an InputSource, an InputStream, or a URI. An example of how to parse an XML document from a File object is as follows:

```
Document document=documentBuilder.parse(new File("catalog.xml"));
```

The Document interface provides various methods to navigate the DOM structure. Table 2-6 lists some of the Document interface methods.

Table 2-6. *Document Interface Methods*

Method Name	Description
getDoctype()	Returns the DOCTYPE in the XML document
getDocumentElement()	Returns the root element
getElementById(String)	Gets an element for a specified ID
getElementsByTagName(String)	Gets a NodeList of elements

The org.w3c.dom.Element interface represents an element in the DOM structure. You can obtain element attributes and subelements from an Element object. Table 2-7 lists some of the methods in the Element interface.

Table 2-7. *Element Interface Methods*

Method Name	Description
getAttributes()	Returns a NamedNodeMap of attributes
getAttribute(String)	Gets an attribute value by attribute name
getAttributeNode(String)	Returns an Attr node for an attribute
getElementsByTagName(String)	Returns a NodeList of elements by element name
getTagName()	Gets the element tag name

The Attr interface represents an attribute node. You can obtain the attribute name and value from the Attr node. Table 2-8 lists some of the methods in the Attr node.

Table 2-8. *Attr Interface Methods*

Method Name	Description
getName()	Returns the attribute name
getValue()	Returns the attribute value

All the specialized Node interfaces, such as Document, Element and Attr, inherit methods defined by the Node interface. Table 2-9 lists some of the methods in the Node interface.

Table 2-9. *Node Interface Methods*

Method Name	Description
getAttributes()	Returns a NamedNodeMap of attributes for an element node
getChildNodes()	Returns the child nodes in a node
getLocalName()	Returns the local name from an element node and an attribute node
getNodeName()	Returns the node name
getNodeValue()	Returns the node value
getNodeType()	Returns the node type

In the example DOM application, retrieve the root element with the getDocumentElement() method, and obtain the root element name with the getTagName() method, as shown in Listing 2-5.

Listing 2-5. *Retrieving the Root Element Name*

```
Element rootElement = document.getDocumentElement();
String rootElementName = rootElement.getTagName();
```

If the root element has attributes, retrieve the attributes in the root element. The hasAttributes() method tests whether an element has attributes, and the getAttributes() method retrieves the attributes, as shown in Listing 2-6.

Listing 2-6. *Retrieving Root Element Attributes*

```
if (rootElement.hasAttributes()) {
  NamedNodeMap attributes = rootElement.getAttributes();
}
```

The getAttributes() method returns a NamedNodeMap of attributes. The NamedNodeMap method getNodeLength() returns the attribute list length, and the attributes in the attribute list are retrieved with the item(int) method. A NamedNodeMap may be iterated over to retrieve the value of attributes, as shown in Listing 2-7. The Attr object method getName() returns the attribute name, and the method getValue() returns the attribute value.

Listing 2-7. *Retrieving Attribute Values*

```
for (int i = 0; i < attributes.getLength(); i++) {
  Attr attribute = (Attr) (attributes.item(i));
  System.out.println("Attribute:" + attribute.getName()+
     " with value " + attribute.getValue());
}
```

If the root element has subnodes, you can retrieve the nodes with the getChildNodes() method. The hasChildNodes() method tests whether an element has subnodes, as shown in Listing 2-8.

Listing 2-8. *Retrieving Nodes in the Root Element*

```
if (rootElement.hasChildNodes()) {
  NodeList nodeList = rootElement.getChildNodes();
}
```

The node list includes whitespace text nodes. The NodeList method getNodeLength() returns the node list length, and you can retrieve the nodes in the node list with the item(int) method, as shown in Listing 2-9.

Listing 2-9. *Retrieving Nodes in a NodeList*

```
for (int i = 0; i < nodeList.getLength(); i++) {
  Node node = nodeList.item(i);
}
```

If a node is of type Element, a Node object may be cast to Element. The node type is obtained with the Node interface method getNodeType(). The getNodeType() method returns a short value. Table 2-10 lists the different short values and the corresponding node types.

Table 2-10. *Node Types*

Short Value	Node Type
ELEMENT_NODE	Element node
ATTRIBUTE_NODE	Attr node
TEXT_NODE	Text node
CDATA_SECTION_NODE	CDATASection node
ENTITY_REFERENCE_NODE	EntityReference node
ENTITY_NODE	Entity node
PROCESSING_INSTRUCTION_NODE	ProcessingInstruction node
COMMENT_NODE	Comment node
DOCUMENT_NODE	Document node
DOCUMENT_TYPE_NODE	DocumentType node
DOCUMENT_FRAGMENT_NODE	DocumentFragment node
NOTATION_NODE	Notation node

If a node is of type Element, cast the Node object to Element, as shown in Listing 2-10.

Listing 2-10. *Casting Node to Element*

```
if (node.getNodeType() == Node.ELEMENT_NODE) {
  Element element = (Element) (node);
}
```

If an element has a text node, you can obtain the text value with the `getNodeValue()` method, as shown here:

```
String textValue=node.getNodeValue();
```

DOM API Example

The Java application `DOMParser.java` shown in Listing 2-11 parses the XML document shown in Listing 2-2. We are assuming you have imported the XML document shown in Listing 2-2 to the `Chapter2` project, as shown in Figure 2-2.

This example demonstrates how to use a `DocumentBuilder` object to parse the example XML document. Once you successfully parse the document, you get a `Document` object, which represents an in-memory tree structure for the example document. You retrieve the node representing the root element from the `Document` object, and you use the `visitNode()` method to walk down this tree and visit each node, starting at the root element.

When you get to a node while traversing the tree, you first find its node type. If the node type is `Element`, you traverse the child nodes of the `Element` node with the `visitNode()` method. The `visitNode()` method also outputs the element tag name and attributes in an element. If the node type is `Text` and the `Text` node is not an empty node, the text value of the `Text` node is output.

Listing 2-11. *DOM Parsing Application `DOMParser.java`*

```
package com.apress.dom;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import java.io.*;

public class DOMParser {

   public static void main(String argv[]) {
      try {
         // Create a DocumentBuilderFactory
         DocumentBuilderFactory factory = DocumentBuilderFactory
               .newInstance();
         File xmlFile = new File("catalog.xml");
         // Create a DocumentBuilder
         DocumentBuilder builder = factory.newDocumentBuilder();
         // Parse an XML document
         Document document = builder.parse(xmlFile);
         // Retrieve root element
         Element rootElement = document.getDocumentElement();
         System.out.println("Root Element is: " + rootElement.getTagName());
         visitNode(null, rootElement);

      } catch (SAXException e) {
         System.out.println(e.getMessage());
      } catch (ParserConfigurationException e) {
         System.out.println(e.getMessage());

      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }

   public static void visitNode(Element previousNode, Element visitNode) {
      // process an Element node
      if (previousNode != null) {
         System.out.println("Element " + previousNode.getTagName()
               + " has element:");
      }
      System.out.println("Element Name: " + visitNode.getTagName());
      // list attributes for an element node
      if (visitNode.hasAttributes()) {
         System.out.println("Element " + visitNode.getTagName()
               + " has attributes: ");
         NamedNodeMap attributes = visitNode.getAttributes();

         for (int j = 0; j < attributes.getLength(); j++) {
            Attr attribute = (Attr) (attributes.item(j));
            System.out.println("Attribute:" + attribute.getName()
                  + " with value " + attribute.getValue());

         }
      }
      // Obtain a NodeList of nodes in an Element node

      NodeList nodeList = visitNode.getChildNodes();
      for (int i = 0; i < nodeList.getLength(); i++) {
         Node node = nodeList.item(i);
         // Retrieve Element nodes
         if (node.getNodeType() == Node.ELEMENT_NODE) {
            Element element = (Element) node;
            // Recursive call to visitNode method to process
            // an Element node hierarchy
            visitNode(visitNode, element);
         } else if (node.getNodeType() == Node.TEXT_NODE) {
            String str = node.getNodeValue().trim();
            if (str.length() > 0) {
               System.out.println("Element Text: " + str);
            }
         }
      }
   }
}
```

Listing 2-12 shows the output from running the DOM application in Eclipse. This output shows the node type and node value associated with each node visited in the tree walk.

Listing 2-12. *Output from the* `DOMParser` *Application*

```
Root Element is: catalog
Element Name: catalog
Element catalog has attributes:
Attribute:publisher with value O'Reilly
Attribute:title with value OnJava.com
Element catalog has element:
Element Name: journal
Element journal has attributes:
Attribute:date with value January 2004
Element journal has element:
Element Name: article
Element article has element:
Element Name: title
Element Text: Data Binding with XMLBeans
Element article has element:
Element Name: author
Element Text: Daniel Steinberg
Element catalog has element:
Element Name: journal
Element journal has attributes:
Attribute:date with value Sept 2005
Element journal has element:
Element Name: article
Element article has element:
Element Name: title
Element Text: What Is Hibernate
Element article has element:
Element Name: author
Element Text: James Elliott
```

Parsing with SAX 2.0

SAX 2.0[5] is an event-based API to parse an XML document. SAX 2.0 is not a W3C Recommendation. However, it is a widely used API that has become a de facto standard. To date, SAX has two major versions: SAX 1.0 and SAX 2.0. There are no fundamental differences between the two versions. The most notable difference is that the SAX 1.0 `Parser` interface is replaced with the SAX 2.0 `XMLReader` interface, which improves upon the SAX 1.0 interface by providing full support for namespaces. In this chapter, we will focus only on the SAX 2.0 API.

SAX 2.0 is a push-model API; events are generated as an XML document is parsed. Events are generated by the parser and delivered through the callback methods defined by the application. Key points pertaining to the use of the SAX 2.0 API are as follows:

- You need to import at least two packages: the `org.xml.sax` package for the SAX interfaces and the `javax.xml.parsers` package for the `SAXParser` and `SAXParserFactory` classes. In addition, you may need to import the `org.xml.sax.helpers` package, which has useful helper classes for using the SAX API.

[5] You can find information about SAX at `http://www.saxproject.org/`.

- ContentHandler is the main interface that an application needs to implement because it provides event notification about the parsing events. The DefaultHandler class provides a default implementation of the ContentHandler interface. To handle SAX parser events, an application can either define a class that implements the ContentHandler interface or define a class that extends the DefaultHandler class.
- You use the SAXParser class to parse an XML document.
- You obtain a SAXParser object from a SAXParserFactory object. To obtain a SAX parser, you need to first create an instance of the SAXParserFactory using the static method newInstance(), as shown in the following example:

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

JAXP Pluggability for SAX

JAXP 1.3 provides complete pluggability for the SAXParserFactory implementation classes. This means the SAXParserFactory implementation class is not a fixed class. Instead, the SAXParserFactory implementation class is obtained by JAXP, using the following lookup procedure:

1. Use the javax.xml.parsers.SAXParserFactory system property to determine the factory class to load.
2. Use the javax.xml.parsers.SAXParserFactory property specified in the lib/jaxp.properties file under the JRE directory to determine the factory class to load. JAXP reads this file only once, and the property values defined in this file are cached by JAXP.
3. Files in the META-INF/services directory within a JAR file are deemed service provider configuration files. Use the Services API, and obtain the factory class name from the META-INF/services/javax.xml.parsers.SAXParserFactory file contained in any JAR file in the runtime classpath.
4. Use the default SAXParserFactory class, included in the J2SE platform.

If validation is desired, set the validating attribute on factory to true:

```
factory.setValidating(true);
```

If the validation attribute of the SAXParserFactory object is set to true, the parser obtained from such a factory object, by default, validates an XML document with respect to a DTD. To validate the document with respect to XML Schema, you need to do more, which is covered in detail in Chapter 3.

SAX Features

SAXParserFactory features are logical switches that you can turn on and off to change parser behavior. You can set the features of a factory through the setFeature(String, boolean) method. The first argument passed to setFeature is the name of a feature, and the second argument is a true or false value. Table 2-11 lists some of the commonly used SAXParserFactory features. Some of the SAXParserFactory features are implementation specific, so not all features may be supported by different factory implementations.

Table 2-11. *SAXParserFactory Features*

Feature	Description
http://xml.org/sax/features/namespaces	Performs namespace processing if set to true
http://xml.org/sax/features/validation	Validates an XML document
http://apache.org/xml/features/validation/schema	Performs XML Schema validation
http://xml.org/sax/features/external-general-entities	Includes external general entities
http://xml.org/sax/features/external-parameter-entities	Includes external parameter entities and the external DTD subset
http://apache.org/xml/features/nonvalidating/load-external-dtd	Loads the external DTD
http://xml.org/sax/features/namespace-prefixes	Reports attributes and prefixes used for namespace declarations
http://xml.org/sax/features/xml-1.1	Supports XML 1.1

SAX Properties

SAX parser properties are name-value pairs that you can use to supply object values to a SAX parser. These properties affect parser behavior and can be set on a parser through the `setProperty(String, Object)` method. The first argument passed to `setProperty` is the name of a property, and the second argument is an `Object` value. Table 2-12 lists some of the commonly used SAX parser properties. Some of the properties are implementation specific, so not all properties may be supported by different SAX parser implementations.

Table 2-12. *SAX Parser Properties*

Property	Description
http://apache.org/xml/properties/schema/external-schemaLocation	Specifies the external schemas for validation
http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation	Specifies external no-namespace schemas
http://xml.org/sax/properties/declaration-handler	Specifies the handler for DTD declarations
http://xml.org/sax/properties/lexical-handler	Specifies the handler for lexical parsing events
http://xml.org/sax/properties/dom-node	Specifies the DOM node being parsed if SAX is used as a DOM iterator
http://xml.org/sax/properties/document-xml-version	Specifies the XML version of the document

SAX Handlers

To parse a document using the SAX 2.0 API, you must define two classes:

- A class that implements the ContentHandler interface (Table 2-2)
- A class that implements the ErrorHandler interface (Table 2-3)

The SAX 2.0 API provides a DefaultHandler helper class that fully implements the ContentHandler and ErrorHandler interfaces and provides default behavior for every parser event type along with default error handling. Applications can extend the DefaultHandler class and override relevant base class methods to implement their custom callback handler. CustomSAXHandler, shown in Listing 2-13, is such a class that overrides some of the base class event notification methods, including the error-handling methods.

Key points about CustomSAXHandler class are as follows:

- In the CustomSAXHandler class, in the startDocument() and endDocument() methods, the event type is output.
- In the startElement() method, the event type, element qualified name, and element attributes are output. The uri parameter of the startElement() method is the namespace uri, which may be null, for an element. The parameter localName is the element name without the element prefix. The parameter qName is the element name with the prefix. If an element is not in a namespace with a prefix, localName is the same as qName.
- The parameter attributes is a list of element attributes. The startElement() method prints the qualified element name and the element attributes. The Attributes interface method getQName() returns the qualified name of an attribute. The attribute method getValue() returns the attribute value.
- The characters() method, which gets invoked for a text event, such as element text, prints the text for a node.
- The three error handler methods—fatalError, error, and warning—print the error messages contained in the SAXParseException object passed to these methods.

Listing 2-13. *CustomSAXHandler Class*

```
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
private class CustomSAXHandler extends DefaultHandler {
      public CustomSAXHandler() {
      }

      public void startDocument() throws SAXException {
                                 //Output Event Type
         System.out.println("Event Type: Start Document");
      }

      public void endDocument() throws SAXException {
                                //Output Event Type
         System.out.println("Event Type: End Document");
      }

      public void startElement(String uri, String localName, String qName,
            Attributes attributes) throws SAXException {
                                 //Output Event Type and Element Name
         System.out.println("Event Type: Start Element");
         System.out.println("Element Name:" + qName);
                                        //Output Element Attributes
         for (int i = 0; i < attributes.getLength(); i++) {
            System.out.println("Attribute Name:" + attributes.getQName(i));
            System.out.println("Attribute Value:" + attributes.getValue(i));
         }

      }

      public void endElement(String uri, String localName, String qName)
            throws SAXException {
                                        //Output Event Type
         System.out.println("Event Type: End Element");
      }

      public void characters(char[] ch, int start, int length)
            throws SAXException {
                                       //Output Event Type and Text
         System.out.println("Event Type:  Text");
         String str = (new String(ch, start, length));
         System.out.println(str);
      }

//Error Handling
       public void error(SAXParseException e)
        throws SAXException{
         System.out.println("Error: "+e.getMessage());
      }

        public void fatalError(SAXParseException e)
             throws SAXException{
           System.out.println("Fatal Error: "+e.getMessage());
        }

        public void warning(SAXParseException e)
          throws SAXException{
           System.out.println("Warning: "+e.getMessage());
        }
   }
```

SAX Parsing Steps

The SAX parsing steps are as follows:

1. Create a SAXParserFactory object with the static method newInstance().
2. Create a SAXParser object from the SAXParserFactory object with the newSAXParser() method.
3. Create a DefaultHandler object, and parse the example XML document with the SAXParser method parse(File, DefaultHandler).

Listing 2-14 shows a code sequence for creating a SAX parser that uses an instance of the CustomSAXHandler class to process SAX events.

Listing 2-14. *Creating a SAX Parser*

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

// create a parser
SAXParser saxParser=factory.newSAXParser();

// create and set event handler on the parser
DefaultHandler handler=new CustomSAXHandler();
saxParser.parse(new File("catalog.xml"), handler);
```

SAX API Example

The parsing events are notified through the DefaultHandler callback methods. The CustomSAXHandler class extends the DefaultHandler class and overrides some of the event notification methods. The CustomSAXHandler class also overrides the error handler methods to perform application-specific error handling. The CustomSAXHandler class is defined as a private class within the SAX parsing application, SAXParserApp.java, as shown in Listing 2-15.

Listing 2-15. *SAXParserApp.java*

```
package com.apress.sax;

import org.xml.sax.*;
import javax.xml.parsers.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;

public class SAXParserApp {

   public static void main(String argv[]) {

      SAXParserApp saxParserApp = new SAXParserApp();
      saxParserApp.parseDocument();

   }

   public void parseDocument() {

      try {        //Create a SAXParserFactory
         SAXParserFactory factory = SAXParserFactory.newInstance();
                                       //Create a SAXParser
         SAXParser saxParser = factory.newSAXParser();
         //Create a DefaultHandler and parser an XML document
         DefaultHandler handler = new CustomSAXHandler();
         saxParser.parse(new File("catalog.xml"), handler);
      } catch (SAXException e) {
      } catch (ParserConfigurationException e) {
      } catch (IOException e) {
      }
   }
```

```
            //DefaultHandler class
 private class CustomSAXHandler extends DefaultHandler {
    public CustomSAXHandler() {
    }

    public void startDocument() throws SAXException {
       System.out.println("Event Type: Start Document");
    }

    public void endDocument() throws SAXException {
       System.out.println("Event Type: End Document");
    }

    public void startElement(String uri, String localName, String qName,
          Attributes attributes) throws SAXException {
       System.out.println("Event Type: Start Element");
       System.out.println("Element Name:" + qName);
       for (int i = 0; i < attributes.getLength(); i++) {
          System.out.println("Attribute Name:" + attributes.getQName(i));
          System.out.println("Attribute Value:" + attributes.getValue(i));
       }

    }

    public void endElement(String uri, String localName, String qName)
          throws SAXException {
       System.out.println("Event Type: End Element");
    }

    public void characters(char[] ch, int start, int length)
          throws SAXException {
       System.out.println("Event Type:  Text");
       String str = (new String(ch, start, length));
       System.out.println(str);
    }

    public void error(SAXParseException e)
      throws SAXException{
       System.out.println("Error "+e.getMessage());
    }

      public void fatalError(SAXParseException e)
           throws SAXException{
         System.out.println("Fatal Error "+e.getMessage());
      }

      public void warning(SAXParseException e)
        throws SAXException{
         System.out.println("Warning "+e.getMessage());
      }
 }

}
```

Listing 2-16 shows the output from SAXParserApp.java. Whitespace between elements is also output as text, because unlike in the case of the DOM API example, the SAX example does not filter out whitespace text.

Listing 2-16. *Output from the SAXParserApp Application*

```
Event Type: Start Document
Event Type: Start Element
Element Name:catalog
Attribute Name:title
Attribute Value:OnJava.com
Attribute Name:publisher
Attribute Value:O'Reilly
Event Type:  Text

Event Type:  Text


Event Type: Start Element
Element Name:journal
Attribute Name:date
Attribute Value:January 2004
Event Type:  Text


Event Type: Start Element
Element Name:article
Event Type:  Text

Event Type:  Text


Event Type: Start Element
Element Name:title
Event Type:  Text
Data Binding with XMLBeans
Event Type: End Element
Event Type:  Text


Event Type: Start Element
Element Name:author
Event Type:  Text
Daniel Steinberg
Event Type: End Element
Event Type:  Text


Event Type: End Element
Event Type:  Text
Event Type: End Element
Event Type:  Text


Event Type: Start Element
Element Name:journal
Attribute Name:date
Attribute Value:Sept 2005
Event Type:  Text

Event Type:  Text


Event Type: Start Element
Element Name:article
Event Type:  Text


Event Type: Start Element
Element Name:title
Event Type:  Text
What Is Hibernate
Event Type: End Element
Event Type:  Text


Event Type: Start Element
Element Name:author
Event Type:  Text
James Elliott
Event Type: End Element
Event Type:  Text


Event Type: End Element
Event Type:  Text


Event Type: End Element
Event Type:  Text


Event Type:  Text


Event Type: End Element
Event Type: End Document
```

To demonstrate error handling in a SAX parsing application, add an error in the example XML document, `catalog.xml`; remove a `</journal>` tag, for example. The SAX parsing application outputs the error in the XML document, as shown in Listing 2-17.

Listing 2-17. *SAX Parsing Error*

```
Fatal Error: The element type
   "journal" must be terminated by the matching end-tag "</journal>".
```

Parsing with StAX

StAX is a pull-model API for parsing XML. StAX has an advantage over the push-model SAX. In the push model, the parser generates events as the XML document is parsed. With the pull parsing in StAX, the application generates the parse events; thus, you can generate parse events as required. The StAX API (JSR-173)[6] is implemented in J2SE 6.0.

Key points about StAX API are as follows:

- The StAX API classes are in the `javax.xml.stream` and `javax.xml.stream.events` packages.
- The StAX API offers two different APIs for parsing an XML document: a cursor-based API and an iterator-based API.
- The `XMLStreamReader` interface parses an XML document using the cursor API.
- `XMLEventReader` parses an XML document using the iterator API.
- You can use the `XMLStreamWriter` interface to generate an XML document.

We will first discuss the cursor API and then the iterator API.

Cursor API

You can use the `XMLStreamReader` object to parse an XML document using the cursor approach. The `next()` method generates the next parse event. You can obtain the event type from the `getEventType()` method. You can create an `XMLStreamReader` object from an `XMLInputFactory` object, and you can create an `XMLInputFactory` object using the static method `newInstance()`, as shown in Listing 2-18.

Listing 2-18. *Creating an `XMLStreamReader` Object*

```
XMLInputFactory inputFactory=XMLInputFactory.newInstance();
InputStream input=new FileInputStream(new File("catalog.xml"));
XMLStreamReader  xmlStreamReader = inputFactory.createXMLStreamReader(input);
```

The next parsing event is generated with the `next()` method of an `XMLStreamReader` object, as shown in Listing 2-19.

Listing 2-19. *Obtaining a Parsing Event*

```
while (xmlStreamReader.hasNext()) {
   int event = xmlStreamReader.next();
}
```

The `next()` method returns an `int`, which corresponds to a parsing event, as specified by an `XMLStreamConstants` constant. Table 2-13 lists the event types returned by the `XMLStreamReader` object.

For a `START_DOCUMENT` event type, the `getEncoding()` method returns the encoding in the XML document. The `getVersion()` method returns the XML document version.

[6] You can find this specification at `http://jcp.org/aboutJava/communityprocess/final/jsr173/index.html`.

Table 2-13. *XMLStreamReader Events*

Event Type	Description
START_DOCUMENT	Start of a document
START_ELEMENT	Start of an element
ATTRIBUTE	An element attribute
NAMESPACE	A namespace declaration
CHARACTERS	Characters may be text or whitespace
COMMENT	A comment
SPACE	Ignorable whitespace
PROCESSING_INSTRUCTION	Processing instruction
DTD	A DTD
ENTITY_REFERENCE	An entity reference
CDATA	CDATA section
END_ELEMENT	End element
END_DOCUMENT	End document
ENTITY_DECLARATION	An entity declaration
NOTATION_DECLARATION	A notation declaration

For a START_ELEMENT event type, the getPrefix() method returns the element prefix, and the getNamespaceURI() method returns the namespace or the default namespace. The getLocalName() method returns the local name of an element, as shown in Listing 2-20.

Listing 2-20. *Outputting the Element Name*

```
if (event == XMLStreamConstants.START_ELEMENT) {
System.out.println("Element Local Name:"+ xmlStreamReader.getLocalName());
}
```

The getAttributesCount() method returns the number of attributes in an element. The getAttributePrefix(int) method returns the attribute prefix for a specified attribute index. The getAttributeNamespace(int) method returns the attribute namespace for a specified attribute index. The getAttributeLocalName(int) method returns the local name of an attribute, and the getAttributeValue(int) method returns the attribute value. The attribute name and value are output as shown in Listing 2-21.

Listing 2-21. *Outputting the Attribute Name and Value*

```
for (int i = 0; i < xmlStreamReader.getAttributeCount(); i++) {
    //Output Attribute Name
   System.out.println("Attribute Local Name:"+
      xmlStreamReader.getAttributeLocalName(i));
   //Output Attribute Value
   System.out.println("Attribute Value:"+ xmlStreamReader.getAttributeValue(i));
}
```

The getText() method retrieves the text of a CHARACTERS event, as shown in Listing 2-22.

Listing 2-22. *Outputting Text*

```
if (event == XMLStreamConstants.CHARACTERS) {
                System.out.println("Text:" + xmlStreamReader.getText());
}
```

Listing 2-23 shows the complete StAX cursor API parsing application.

Listing 2-23. *StAXParser.java*

```
package com.apress.stax;

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

public class StAXParser {

   public void parseXMLDocument () {
      try {
         //Create XMLInputFactory object
         XMLInputFactory inputFactory = XMLInputFactory.newInstance();
          //Create XMLStreamReader
         InputStream input = new FileInputStream(new File("catalog.xml"));
         XMLStreamReader xmlStreamReader = inputFactory
              .createXMLStreamReader(input);
         //Obtain StAX Parsing Events
         while (xmlStreamReader.hasNext()) {
            int event = xmlStreamReader.next();

            if (event == XMLStreamConstants.START_DOCUMENT) {
            System.out.println("Event Type:START_DOCUMENT");
            }
            if (event == XMLStreamConstants.START_ELEMENT) {
               System.out.println("Event Type: START_ELEMENT");
               //Output Element Local Name
               System.out.println("Element Local Name:"
                     + xmlStreamReader.getLocalName());
               //Output Element Attributes
            for (int i = 0; i < xmlStreamReader.getAttributeCount(); i++) {

                        System.out.println("Attribute Local Name:"
                  + xmlStreamReader.getAttributeLocalName(i));
               System.out.println("Attribute Value:"
                  + xmlStreamReader.getAttributeValue(i));
               }

            }
            if (event == XMLStreamConstants.CHARACTERS) {
               System.out.println("Event Type: CHARACTERS");
            System.out.println("Text:" + xmlStreamReader.getText());
            }

            if (event == XMLStreamConstants.END_DOCUMENT) {
               System.out.println("Event Type:END_DOCUMENT");
            }
            if (event == XMLStreamConstants.END_ELEMENT) {
               System.out.println("Event Type: END_ELEMENT");
            }

         }
      } catch (FactoryConfigurationError e) {
         System.out.println("FactoryConfigurationError" + e.getMessage());
      } catch (XMLStreamException e) {
         System.out.println("XMLStreamException" + e.getMessage());
      } catch (IOException e) {
         System.out.println("IOException" + e.getMessage());
      }

   }

   public static void main(String[] argv) {

      StAXParser staxParser = new StAXParser();
      staxParser.parseXMLDocument();

   }
}
```

Listing 2-24 shows the output from the StAX parsing application in Eclipse.

Listing 2-24. *Output from the* `StAXParser` *Application*

```
Event Type: START_ELEMENT
Element Local Name:catalog
Attribute Local Name:title
Attribute Value:OnJava.com
Attribute Local Name:publisher
Attribute Value:O'Reilly
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:journal
Attribute Local Name:date
Attribute Value:January 2004
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:article
Event Type: CHARACTERS
Text:
```

```
Event Type: START_ELEMENT
Element Local Name:title
Event Type: CHARACTERS
Text:Data Binding with XMLBeans
Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:author
Event Type: CHARACTERS
Text:Daniel Steinberg
Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:journal
Attribute Local Name:date
Attribute Value:Sept 2005
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:article
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:title
Event Type: CHARACTERS
Text:What Is Hibernate
Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: START_ELEMENT
Element Local Name:author
Event Type: CHARACTERS
Text:James Elliott
Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:
```

```
Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:

Event Type: END_ELEMENT
Event Type: CHARACTERS
Text:


Event Type: END_ELEMENT
Event Type:END_DOCUMENT
```

Iterator API

The XMLEventReader object parses an XML document with an object event iterator and generates an XMLEvent object for each parse event. To create an XMLEventReader object, you need to first create an XMLInputFactory object with the static method newInstance() and then obtain an XMLEventReader object from the XMLInputFactory object with the createXMLEventReader method, as shown in Listing 2-25.

Listing 2-25. *Creating an XMLEventReader Object*

```
XMLInputFactory inputFactory=XMLInputFactory.newInstance();
InputStream input=new FileInputStream(new File("catalog.xml"));
XMLEventReader  xmlEventReader  = inputFactory.createXMLEventReader(input);
```

An XMLEvent object represents an XML document event in StAX. You obtain the next event with the nextEvent() method of an XMLEventReader object. The getEventType() method of an XMLEventReader object returns the event type, as shown here:

```
XMLEvent event=xmlEventReader.nextEvent();
int eventType=event.getEventType();
```

The event types listed in Table 2-13 for an XMLStreamReader object are also the event types generated with an XMLEventReader object. The isXXX() methods in the XMLEventReader interface return a boolean if the event is of the type corresponding to the isXXX() method. For example, the isStartDocument() method returns true if the event is of type START_DOCUMENT. You can use relevant XMLStreamReader methods to process event types that are of interest to the application.

Summary

You can parse an XML document using one of three methods: DOM, push, or pull.

The DOM approach provides random access and a complete ability to manipulate document elements and attributes; however, this approach consumes the most memory. This approach is best for use in situations where an in-memory model of the XML structure and content is required so that an application can easily manipulate the structure and content of an XML document. Applications that need to visualize an XML document and manipulate the document through a user interface may find this API extremely relevant to their application objectives. The DOM Level 3 API included in JAXP 1.3 implements this approach.

The push approach is based on a simple event notification model where a parser synchronously delivers parsing events so an application can handle these events by implementing a callback handler interface. The SAX 2.0 API is best suited for situations where the core objectives are as follows: quickly parse an XML document, make sure it is well-formed and valid, and extract content information contained in the document as the document is being parsed. It is worth noting that a DOM API implementation could internally use a SAX 2.0 API–based parser to parse an XML document and build a DOM tree, but it is not required to do so. The SAX 2.0 API included in JAXP 1.3 implements this approach.

The pull approach provides complete control to an application over how the document parse events are processed and provides a cursor-based approach and an iterator-based approach to control the flow of parse events. This approach is best suited for processing XML content that is being streamed over a network connection. Also, this API is useful for marshaling and unmarshaling XML documents from and to Java types. Major areas of applications for this API include web services–related message processing and XML-to-Java binding. The StAX API included in J2SE 6.0 implements this approach.