Lesson: Simple API for XML

This lesson focuses on the Simple API for XML (SAX), an event-driven, serial-access mechanism for accessing XML documents. This protocol is frequently used by servlets and network-oriented programs that need to transmit and receive XML documents, because it is the fastest and least memory-intensive mechanism that is currently available for dealing with XML documents, other than the Streaming API for XML (StAX).

Note - In a nutshell, SAX is oriented towards state independent processing, where the handling of an element does not depend on the elements that came before. StAX, on the other hand, is oriented towards state dependent processing. For a more detailed comparison, see When to Use SAX.

Setting up a program to use SAX requires a bit more work than setting up to use the Document Object Model (DOM). SAX is an event-driven model (you provide the callback methods, and the parser invokes them as it reads the XML data), and that makes it harder to visualize. Finally, you cannot "back up" to an earlier part of the document, or rearrange it, any more than you can back up a serial data stream or rearrange characters you have read from that stream.

For those reasons, developers who are writing a user-oriented application that displays an XML document and possibly modifies it will want to use the DOM mechanism described in Document Object Model.

However, even if you plan to build DOM applications exclusively, there are several important reasons for familiarizing yourself with the SAX model:

- **Same Error Handling**: The same kinds of exceptions are generated by the SAX and DOM APIs, so the error handling code is virtually identical.
- **Handling Validation Errors**: By default, the specifications require that validation errors be ignored. If you want to throw an exception in the event of a validation error (and you probably do), then you need to understand how SAX error handling works.
- **Converting Existing Data**: As you will see in Document Object Model, there is a mechanism you can use to convert an existing data set to XML. However, taking advantage of that mechanism requires an understanding of the SAX model.

When to Use SAX

It is helpful to understand the SAX event model when you want to convert existing data to XML. The key to the conversion process is to modify an existing application to deliver SAX events as it reads the data.

SAX is fast and efficient, but its event model makes it most useful for such state-independent filtering. For example, a SAX parser calls one method in your application when an element tag is encountered and calls a different method when text is found. If the processing you are doing is state-independent (meaning that it does not depend on the elements that have come before), then SAX works fine.

On the other hand, for state-dependent processing, where the program needs to do one thing with the data under element A but something different with the data under element B, then a pull parser such as the Streaming API for XML (StAX) would be a better choice. With a pull parser, you get the next node, whatever it happens to be, at any point in the code that you ask for it. So it is easy to vary the way you process text (for example), because you can process it multiple places in the program (for more detail, see Further Information).

SAX requires much less memory than DOM, because SAX does not construct an internal representation (tree structure) of the XML data, as a DOM does. Instead, SAX simply sends data to the application as it is read; your application can then do whatever it wants to do with the data it sees.

Pull parsers and the SAX API both act like a serial I/O stream. You see the data as it streams in, but you cannot go back to an earlier position or leap ahead to a different position. In general, such parsers work well when you simply want to read data and have the application act on it.

But when you need to modify an XML structure - especially when you need to modify it interactively - an in-memory structure makes more sense. DOM is one such model. However, although DOM provides many powerful capabilities for large-scale documents (like books and articles), it also requires a lot of complex coding. The details of that process are highlighted in When to Use DOM in the next lesson.

For simpler applications, that complexity may well be unnecessary. For faster development and simpler applications, one of the object-oriented XML-programming standards, such as JDOM (http://www.jdom.org) and DOM4J (http://www.dom4j.org/), might make more sense.

Parsing an XML File Using SAX

In real-life applications, you will want to use the SAX parser to process XML data and do something useful with it. This section examines an example JAXP program, `SAXLocalNameCount`, that counts the number of elements using only the `localName` component of the element, in an XML document. Namespace names are ignored for simplicity. This example also shows how to use a SAX `ErrorHandler`.

Note - After you have downloaded and installed the sources of the JAXP API from the JAXP download area, the sample program for this example is found in the directory *install-dir*`/jaxp-1_4_2-`*release-date*`/samples/sax`. The XML files it interacts with are found in *install-dir*`/jaxp-1_4_2-`*release-date*`/samples/data`.

Creating the Skeleton

The `SAXLocalNameCount` program is created in a file named `SAXLocalNameCount.java`.

```
public class SAXLocalNameCount {
    static public void main(String[] args) {
        // ...
    }
}
```

Because you will run it standalone, you need a `main()` method. And you need command-line arguments so that you can tell the application which file to process.

Importing Classes

The import statements for the classes the application will use are the following.

```
package sax;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

import java.util.*;
import java.io.*;

public class SAXLocalNameCount {
    // ...
}
```

The `javax.xml.parsers` package contains the `SAXParserFactory` class that creates the parser instance used. It throws a `ParserConfigurationException` if it cannot produce a parser that matches the specified configuration of options. (Later, you will see more about the configuration options). The `javax.xml.parsers` package also contains the `SAXParser` class, which is what the factory returns for parsing. The `org.xml.sax` package defines all the interfaces used for the SAX parser. The `org.xml.sax.helpers` package contains `DefaultHandler`, which defines the class that will handle the SAX events that the parser generates. The classes in `java.util` and `java.io`, are needed to provide hash tables and output.

Setting Up I/O

The first order of business is to process the command-line arguments, which at this stage only serve to get the name of the file to process. The following code in the `main` method tells the application what file you want `SAXLocalNameCountMethod` to process.

```
static public void main(String[] args) throws Exception {
    String filename = null;

    for (int i = 0; i < args.length; i++) {
        filename = args[i];
        if (i != args.length - 1) {
            usage();
        }
    }

    if (filename == null) {
        usage();
    }
}
```

This code sets the main method to throw an `Exception` when it encounters problems, and defines the command-line options which are required to tell the application the name of the XML file to be processed. Other command line arguments in this part of the code will be examined later in this lesson, when we start looking at validation.

The `filename` String that you give when you run the application will be converted to a `java.io.File` URL by an internal method, `convertToFileURL()`. This is done by the following code in `SAXLocalNameCountMethod`.

```
public class SAXLocalNameCount {
    private static String convertToFileURL(String filename) {
        String path = new File(filename).getAbsolutePath();
        if (File.separatorChar != '/') {
            path = path.replace(File.separatorChar, '/');
        }

        if (!path.startsWith("/")) {
            path = "/" + path;
        }
        return "file:" + path;
    }

    // ...
}
```

If the incorrect command-line arguments are specified when the program is run, then the `SAXLocalNameCount` application's `usage()` method is invoked, to print out the correct options onscreen.

```
private static void usage() {
    System.err.println("Usage: SAXLocalNameCount <file.xml>");
    System.err.println("       -usage or -help = this message");
    System.exit(1);
}
```

Further `usage()` options will be examined later in this lesson, when validation is addressed.

Implementing the `ContentHandler` Interface

The most important interface in `SAXLocalNameCount` is `ContentHandler`. This interface requires a number of methods that the SAX parser invokes in response to various parsing events. The major event-handling methods are: `startDocument`, `endDocument`, `startElement`, and `endElement`.

The easiest way to implement this interface is to extend the `DefaultHandler` class, defined in the `org.xml.sax.helpers` package. That class provides do-nothing methods for all the `ContentHandler` events. The example program extends that class.

```
public class SAXLocalNameCount extends DefaultHandler {
    // ...
}
```

Note - `DefaultHandler` also defines do-nothing methods for the other major events, defined in the `DTDHandler`, `EntityResolver`, and `ErrorHandler` interfaces. You will learn more about those methods later in this lesson.

Each of these methods is required by the interface to throw a `SAXException`. An exception thrown here is sent back to the parser, which sends it on to the code that invoked the parser.

Handling Content Events

This section shows the code that processes the `ContentHandler` events.

When a start tag or end tag is encountered, the name of the tag is passed as a String to the `startElement` or the `endElement` method, as appropriate. When a start tag is encountered, any attributes it defines are also passed in an `Attributes` list. Characters found within the element are passed as an array of characters, along with the number of characters (length) and an offset into the array that points to the first character.

Document Events

The following code handles the start-document and end-document events:

```
public class SAXLocalNameCount extends DefaultHandler {

    private Hashtable tags;

    public void startDocument() throws SAXException {
        tags = new Hashtable();
    }

    public void endDocument() throws SAXException {
        Enumeration e = tags.keys();
        while (e.hasMoreElements()) {
            String tag = (String)e.nextElement();
            int count = ((Integer)tags.get(tag)).intValue();
            System.out.println("Local Name \"" + tag + "\" occurs "
                               + count + " times");
        }
    }

    private static String convertToFileURL(String filename) {
```

```
        // ...
    }

    // ...
}
```

This code defines what the application does when the parser encounters the start and end points of the document being parsed. The `ContentHandler` interface's `startDocument()` method creates a `java.util.Hashtable` instance, which in Element Events will be populated with the XML elements the parser finds in the document. When the parser reaches the end of the document, the `endDocument()` method is invoked, to get the names and counts of the elements contained in the hash table, and print out a message onscreen to tell the user how many incidences of each element were found.

Both of these `ContentHandler` methods throw `SAXExceptions`. You will learn more about SAX exceptions in Setting up Error Handling.

Element Events

As mentioned in Document Events, the hash table created by the `startDocument` method needs to be populated with the various elements that the parser finds in the document. The following code processes the start-element and end-element events:

```
public void startDocument() throws SAXException {
    tags = new Hashtable();
}

public void startElement(String namespaceURI,
                         String localName,
                         String qName,
                         Attributes atts)
    throws SAXException {

    String key = localName;
    Object value = tags.get(key);

    if (value == null) {
        tags.put(key, new Integer(1));
    }
    else {
        int count = ((Integer)value).intValue();
        count++;
        tags.put(key, new Integer(count));
    }
}

public void endDocument() throws SAXException {
    // ...
}
```

This code processes the element tags, including any attributes defined in the start tag, to obtain the namespace universal resource identifier (URI), the local name and the qualified name of that element. The `startElement()` method then populates the hash map created by `startDocument()` with the local names and the counts thereof, for each type of element. Note that when the `startElement()` method is invoked, if namespace processing is not enabled, then the local name for elements and attributes could turn out to be an empty string. The code handles that case by using the qualified name whenever the simple name is an empty string.

Character Events

The JAXP SAX API also allows you to handle the characters that the parser delivers to your application, using the `ContentHandler.characters()` method.

Note - Character events are not demonstrated in the `SAXLocalNameCount` example, but a brief description is included in this section, for completeness.

Parsers are not required to return any particular number of characters at one time. A parser can return anything from a single character at a time up to several thousand and still be a standard-conforming implementation. So if your application needs to process the characters it sees, it is wise to have the `characters()` method accumulate the characters in a `java.lang.StringBuffer` and operate on them only when you are sure that all of them have been found.

You finish parsing text when an element ends, so you normally perform your character processing at that point. But you might also want to process text when an element starts. This is necessary for document-style data, which can contain XML elements that are intermixed with text. For example, consider this document fragment:

`<para>``This paragraph contains` **`<bold>`**`important`**`</bold>`** `ideas.`**`</para>`**

The initial text, `This paragraph contains`, is terminated by the start of the `<bold>` element. The text `important` is terminated by the end tag, `</bold>`, and the final text, `ideas.`, is terminated by the end tag, `</para>`.

To be strictly accurate, the character handler should scan for ampersand characters (&) and left-angle bracket characters (<) and replace them with the strings `&amp;` or `&lt;`, as appropriate. This is explained in the next section.

Handling Special Characters

In XML, an entity is an XML structure (or plain text) that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, you surround the entity name with an ampersand and a semicolon:

```
&entityName;
```

When you are handling large blocks of XML or HTML that include many special characters, you can use a CDATA section. A CDATA section works like `<code>...</code>` in HTML, only more so: all white space in a CDATA section is significant, and characters in it are not interpreted as XML. A CDATA section starts with `<![[CDATA[` and ends with `]]>`.

An example of a CDATA section, taken from the sample XML file *install-dir*`/jaxp-1_4_2-`*release-date*`/samples/data/REC-xml-19980210.xml`, is shown below.

```
<p><termdef id="dt-cdsection" term="CDATA Section"<<term>CDATA
sections</term> may occur anywhere character data may occur; they
are used to escape blocks of text containing characters which
would otherwise be recognized as markup. CDATA sections begin with
the string "<code>&lt;![CDATA[</code>" and end with the string
"<code>]]&gt;</code>"
```

Once parsed, this text would be displayed as follows:

CDATA sections may occur anywhere character data may occur; they are used to escape blocks of

text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "<![CDATA[" and end with the string "]]>".

The existence of CDATA makes the proper echoing of XML a bit tricky. If the text to be output is not in a CDATA section, then any angle brackets, ampersands, and other special characters in the text should be replaced with the appropriate entity reference. (Replacing left angle brackets and ampersands is most important, other characters will be interpreted properly without misleading the parser.) But if the output text is in a CDATA section, then the substitutions should not occur, resulting in text like that in the earlier example. In a simple program such as our `SAXLocalNameCount` application, this is not particularly serious. But many XML-filtering applications will want to keep track of whether the text appears in a CDATA section, so that they can treat special characters properly.

Setting up the Parser

The following code sets up the parser and gets it started:

```
static public void main(String[] args) throws Exception {

    // Code to parse command-line arguments
    //(shown above)
    // ...

    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setNamespaceAware(true);
    SAXParser saxParser = spf.newSAXParser();
}
```

These lines of code create a `SAXParserFactory` instance, as determined by the setting of the `javax.xml.parsers.SAXParserFactory` system property. The factory to be created is set up to support XML namespaces by setting `setNamespaceAware` to true, and then a `SAXParser` instance is obtained from the factory by invoking its `newSAXParser()` method.

Note - The `javax.xml.parsers.SAXParser` class is a wrapper that defines a number of convenience methods. It wraps the (somewhat less friendly) `org.xml.sax.Parser` object. If needed, you can obtain that parser using the `getParser()` method of the `SAXParser` class.

You now need to implement the `XMLReader` that all parsers must implement. The `XMLReader` is used by the application to tell the SAX parser what processing it is to perform on the document in question. The `XMLReader` is implemented by the following code in the `main` method.

```
// ...
SAXParser saxParser = spf.newSAXParser();
XMLReader xmlReader = saxParser.getXMLReader();
xmlReader.setContentHandler(new SAXLocalNameCount());
xmlReader.parse(convertToFileURL(filename));
```

Here, you obtain an `XMLReader` instance for your parser by invoking your `SAXParser` instance's `getXMLReader()` method. The `XMLReader` then registers the `SAXLocalNameCount` class as its content handler, so that the actions performed by the parser will be those of the `startDocument()`, `startElement()`, and `endDocument()` methods shown in Handling Content Events. Finally, the `XMLReader` tells the parser which document to parse by passing it the location of the XML file in question, in the form of the `File` URL generated by the

`convertToFileURL()` method defined in Setting Up I/O.

Setting up Error Handling

You could start using your parser now, but it is safer to implement some error handling. The parser can generate three kinds of errors: a fatal error, an error, and a warning. When a fatal error occurs, the parser cannot continue. So if the application does not generate an exception, then the default error-event handler generates one. But for nonfatal errors and warnings, exceptions are never generated by the default error handler, and no messages are displayed.

As shown in Document Events, the application's event handling methods throw `SAXException`. For example, the signature of the `startDocument()` method in the `ContentHandler` interface is defined as returning a `SAXException`.

```
public void startDocument() throws SAXException { /* ... */ }
```

A `SAXException` can be constructed using a message, another exception, or both.

Because the default parser only generates exceptions for fatal errors, and because the information about the errors provided by the default parser is somewhat limited, the `SAXLocalNameCount` program defines its own error handling, through the `MyErrorHandler` class.

```
xmlReader.setErrorHandler(new MyErrorHandler(System.err));

// ...

private static class MyErrorHandler implements ErrorHandler {
    private PrintStream out;

    MyErrorHandler(PrintStream out) {
        this.out = out;
    }

    private String getParseExceptionInfo(SAXParseException spe) {
        String systemId = spe.getSystemId();

        if (systemId == null) {
            systemId = "null";
        }

        String info = "URI=" + systemId + " Line="
            + spe.getLineNumber() + ": " + spe.getMessage();

        return info;
    }

    public void warning(SAXParseException spe) throws SAXException {
        out.println("Warning: " + getParseExceptionInfo(spe));
    }

    public void error(SAXParseException spe) throws SAXException {
        String message = "Error: " + getParseExceptionInfo(spe);
        throw new SAXException(message);
    }

    public void fatalError(SAXParseException spe) throws SAXException {
        String message = "Fatal Error: " + getParseExceptionInfo(spe);
        throw new SAXException(message);
    }
}
```

In the same way as in Setting up the Parser, which showed the `XMLReader` being pointed to the correct content handler, here the `XMLReader` is pointed to the new error handler by calling its `setErrorHandler()` method.

The `MyErrorHandler` class implements the standard `org.xml.sax.ErrorHandler` interface, and defines a method to obtain the exception information that is provided by any `SAXParseException` instances generated by the parser. This method, `getParseExceptionInfo()`, simply obtains the line number at which the error occurs in the XML document and the identifier of the system on which it is running by calling the standard `SAXParseException` methods `getLineNumber()` and `getSystemId()`. This exception information is then fed into implementations of the basic SAX error handling methods `error()`, `warning()`, and `fatalError()`, which are updated to send the appropriate messages about the nature and location of the errors in the document.

Handling NonFatal Errors

A nonfatal error occurs when an XML document fails a validity constraint. If the parser finds that the document is not valid, then an error event is generated. Such errors are generated by a validating parser, given a document type definition (DTD) or schema, when a document has an invalid tag, when a tag is found where it is not allowed, or (in the case of a schema) when the element contains invalid data.

The most important principle to understand about nonfatal errors is that they are ignored by default. But if a validation error occurs in a document, you probably do not want to continue processing it. You probably want to treat such errors as fatal.

To take over error handling, you override the `DefaultHandler` methods that handle fatal errors, nonfatal errors, and warnings as part of the `ErrorHandler` interface. As shown in the code extract in the previous section, the SAX parser delivers a `SAXParseException` to each of these methods, so generating an exception when an error occurs is as simple as throwing it back.

Note - It can be instructive to examine the error-handling methods defined in `org.xml.sax.helpers.DefaultHandler`. You will see that the `error()` and `warning()` methods do nothing, whereas `fatalError()` throws an exception. Of course, you could always override the `fatalError()` method to throw a different exception. But if your code does not throw an exception when a fatal error occurs, then the SAX parser will. The XML specification requires it.

Handling Warnings

Warnings, too, are ignored by default. Warnings are informative and can only be generated in the presence of a DTD or schema. For example, if an element is defined twice in a DTD, a warning is generated. It is not illegal, and it does not cause problems, but it is something you might like to know about because it might not have been intentional. Validating an XML document against a DTD will be shown in the section .

Running the SAX Parser Example without Validation

As stated at the beginning of this lesson, after you have downloaded and installed the sources of the JAXP API from the JAXP sources download area, the sample program and the associated files

needed to run it are found in the following locations.

- The different Java archive (JAR) files for the example are located in the directory *install-dir*`/jaxp-1_4_2-`*release-date*`/lib`.
- The `SAXLocalNameCount.java` file is found in *install-dir*`/jaxp-1_4_2-`*release-date*`/samples/sax`.
- The XML files that `SAXLocalNameCount` interacts with are found in *install-dir*`/jaxp-1_4_2-`*release-date*`/samples/data`.

The following steps explain how to run the SAX parser example without validation.

To Run the `SAXLocalNameCount` Example without Validation

1. **Navigate to the `samples` directory.**`% cd` *install-dir*`/jaxp-1_4_2-`*release-date*`/samples.`
2. **Compile the example class.**`% javac sax/*`
3. **Run the `SAXLocalNameCount` program on an XML file.**

   Choose one of the XML files in the `data` directory and run the `SAXLocalNameCount` program on it. Here, we have chosen to run the program on the file `rich_iii.xml`.

   ```
   % java sax/SAXLocalNameCount data/rich_iii.xml
   ```

   The XML file `rich_iii.xml` contains an XML version of William Shakespeare's play *Richard III*. When you run the `SAXLocalNameCount` on it, you should see the following output.

   ```
   Local Name "STAGEDIR" occurs 230 times
   Local Name "PERSONA" occurs 39 times
   Local Name "SPEECH" occurs 1089 times
   Local Name "SCENE" occurs 25 times
   Local Name "ACT" occurs 5 times
   Local Name "PGROUP" occurs 4 times
   Local Name "PLAY" occurs 1 times
   Local Name "PLAYSUBT" occurs 1 times
   Local Name "FM" occurs 1 times
   Local Name "SPEAKER" occurs 1091 times
   Local Name "TITLE" occurs 32 times
   Local Name "GRPDESCR" occurs 4 times
   Local Name "P" occurs 4 times
   Local Name "SCNDESCR" occurs 1 times
   Local Name "PERSONAE" occurs 1 times
   Local Name "LINE" occurs 3696 times
   ```

   The `SAXLocalNameCount` program parses the XML file, and provides a count of the number of instances of each type of XML tag that it contains.

4. **Open the file `data/rich_iii.xml` in a text editor.**

   To check that the error handling is working, delete the closing tag from an entry in the XML file, for example the closing tag `</PERSONA>`, from line 30, shown below.

   ```
   30 <PERSONA>EDWARD, Prince of Wales, afterwards King Edward
   V.</PERSONA>
   ```

5. **Run `SAXLocalNameCount` again.**

   This time, you should see the following fatal error message.

   ```
   % java sax/SAXLocalNameCount data/rich_iii.xml Exception in
   ```

```
thread "main" org.xml.sax.SAXException: Fatal Error:
URI=file:install-dir /JAXP_sources/jaxp-1_4_2-release-
date/samples/data/rich_iii.xml Line=30: The element type
"PERSONA" must be terminated by the matching end-tag
"</PERSONA>".
```

As you can see, when the error was encountered, the parser generated a `SAXParseException`, a subclass of `SAXException` that identifies the file and the location where the error occurred.

Implementing SAX Validation

The sample program `SAXLocalNameCount` uses the non-validating parser by default, but it can also activate validation. Activating validation allows the application to tell whether the XML document contains the right tags or whether those tags are in the right sequence. In other words, it can tell you whether the document is *valid*. If validation is not activated, however, it can only tell whether or not the document is well-formed, as was shown in the previous section when you deleted the closing tag from an XML element. For validation to be possible, the XML document needs to be associated to a DTD or an XML schema. Both options are possible with the `SAXLocalNameCount` program.

Choosing the Parser Implementation

If no other factory class is specified, the default `SAXParserFactory` class is used. To use a parser from a different manufacturer, you can change the value of the environment variable that points to it. You can do that from the command line:

```
java -Djavax.xml.parsers.SAXParserFactory=yourFactoryHere [...]
```

The factory name you specify must be a fully qualified class name (all package prefixes included). For more information, see the documentation in the `newInstance()` method of the `SAXParserFactory` class.

Using the Validating Parser

Up until this point, this lesson has concentrated on the non-validating parser. This section examines the validating parser to find out what happens when you use it to parse the sample program.

Two things must be understood about the validating parser:

- A schema or DTD is required.
- Because the schema or DTD is present, the `ContentHandler.ignorableWhitespace()` method is invoked whenever possible.

Ignorable White Space

When a DTD is present, the parser will no longer call the `characters()` method on white space that it knows to be irrelevant. From the standpoint of an application that is interested in processing only the XML data, that is a good thing because the application is never bothered with white space that exists purely to make the XML file readable.

On the other hand, if you are writing an application that filters an XML data file and if you want to output an equally readable version of the file, then that white space would no longer be irrelevant: it would be essential. To get those characters, you would add the `ignorableWhitespace` method to your application. To process any (generally) ignorable white space that the parser sees, you would need to add something like the following code to implement the `ignorableWhitespace` event handler.

```
public void ignorableWhitespace (char buf[], int start, int
length) throws SAXException { emit("IGNORABLE"); }
```

This code simply generates a message to let you know that ignorable white space was seen. However, not all parsers are created equal. The SAX specification does not require that this method

be invoked. The Java XML implementation does so whenever the DTD makes it possible.

Configuring the Factory

The `SAXParserFactory` needs to be set up such that it uses a validating parser instead of the default non-validating parser. The following code from the `SAXLocalNameCount` example's `main()` method shows how to configure the factory so that it implements the validating parser.

```
static public void main(String[] args) throws Exception {

    String filename = null;
    boolean dtdValidate = false;
    boolean xsdValidate = false;
    String schemaSource = null;

    for (int i = 0; i < args.length; i++) {

        if (args[i].equals("-dtd")) {
            dtdValidate = true;
        }
        else if (args[i].equals("-xsd")) {
            xsdValidate = true;
        }
        else if (args[i].equals("-xsdss")) {
            if (i == args.length - 1) {
               usage();
            }
            xsdValidate = true;
            schemaSource = args[++i];
        }
        else if (args[i].equals("-usage")) {
            usage();
        }
        else if (args[i].equals("-help")) {
            usage();
        }
        else {
            filename = args[i];
            if (i != args.length - 1) {
                usage();
            }
        }
    }

    if (filename == null) {
        usage();
    }

    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setNamespaceAware(true);
    spf.setValidating(dtdValidate || xsdValidate);
    SAXParser saxParser = spf.newSAXParser();

    // ...
}
```

Here, the `SAXLocalNameCount` program is configured to take additional arguments when it is started, which tell it to implement no validation, DTD validation, XML Schema Definition (XSD) validation, or XSD validation against a specific schema source file. (Descriptions of these options, `-dtd`, `-xsd`, and `-xsdss` are also added to the `usage()` method, but are not shown here.) Then, the factory is configured so that it will produce the appropriate validating parser when

`newSAXParser` is invoked. As seen in Setting up the Parser, you can also use `setNamespaceAware(true)` to configure the factory to return a namespace-aware parser. Sun's implementation supports any combination of configuration options. (If a combination is not supported by a particular implementation, it is required to generate a factory configuration error).

Validating with XML Schema

Although a full treatment of XML Schema is beyond the scope of this tutorial, this section shows you the steps you take to validate an XML document using an existing schema written in the XML Schema language. To learn more about XML Schema, you can review the online tutorial, *XML Schema Part 0: Primer*, at http://www.w3.org/TR/xmlschema-0/.

Note - There are multiple schema-definition languages, including RELAX NG, Schematron, and the W3C "XML Schema" standard. (Even a DTD qualifies as a "schema," although it is the only one that does not use XML syntax to describe schema constraints.) However, "XML Schema" presents us with a terminology challenge. Although the phrase "XML Schema schema" would be precise, we will use the phrase "XML Schema definition" to avoid the appearance of redundancy.

To be notified of validation errors in an XML document, the parser factory must be configured to create a validating parser, as shown in the preceding section. In addition, the following must be true:

- The appropriate properties must be set on the SAX parser.
- The appropriate error handler must be set.
- The document must be associated with a schema.

Setting the SAX Parser Properties

It is helpful to start by defining the constants you will use when setting the properties. The `SAXLocalNameCount` example sets the following constants.

```
public class SAXLocalNameCount extends DefaultHandler {

    static final String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

    static final String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

    static final String JAXP_SCHEMA_SOURCE =
        "http://java.sun.com/xml/jaxp/properties/schemaSource";
}
```

Note - The parser factory must be configured to generate a parser that is namespace-aware as well as validating. This was shown in Configuring the Factory. More information about namespaces is provided in Document Object Model but for now, understand that schema validation is a namespace-oriented process. Because JAXP-compliant parsers are not namespace-aware by default, it is necessary to set the property for schema validation to work.

Then you must configure the parser to tell it which schema language to use. In `SAXLocalNameCount`, validation can be performed either against a DTD or against an XML

Schema. The following code uses the constants defined above to specify the W3C's XML Schema language as the one to use if the `-xsd` option is specified when the program is started.

```
// ...
if (xsdValidate) {
    saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
    // ...
}
```

In addition to the error handling described in Setting up Error Handling, there is one error that can occur when configuring the parser for schema-based validation. If the parser is not compliant with JAXP version 1.2 or later, and therefore does not support XML Schema, it can throw a `SAXNotRecognizedException`. To handle that case, the `setProperty()` statement is wrapped in a try/catch block, as shown in the code below.

```
// ...
if (xsdValidate) {
    try {
        saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
    }
    catch (SAXNotRecognizedException x){
        System.err.println("Error: JAXP SAXParser property not recognized: "
                           + JAXP_SCHEMA_LANGUAGE);

        System.err.println( "Check to see if parser conforms to JAXP 1.2
spec.");
        System.exit(1);
    }
}
// ...
```

Associating a Document with a Schema

To validate the data using an XML Schema definition, it is necessary to ensure that the XML document is associated with one. There are two ways to do that.

- By including a schema declaration in the XML document.
- By specifying the schema to use in the application.

Note - When the application specifies the schema to use, it overrides any schema declaration in the document.

To specify the schema definition in the document, you would create XML such as this:

```
<documentRoot
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation='YourSchemaDefinition.xsd'>
```

The first attribute defines the XML namespace (`xmlns`) prefix, `xsi`, which stands for XML Schema instance. The second line specifies the schema to use for elements in the document that do not have a namespace prefix, namely for the elements that are typically defined in any simple, uncomplicated XML document.

Note - More information about namespaces is included in Validating with XML Schema in

Document Object Model. For now, think of these attributes as the "magic incantation" you use to validate a simple XML file that does not use them. After you have learned more about namespaces, you will see how to use XML Schema to validate complex documents that do use them. Those ideas are discussed in Validating with Multiple Namespaces in Document Object Model.

You can also specify the schema file in the application, as is the case in `SAXLocalNameCount`.

```
// ...
if (schemaSource != null) {
    saxParser.setProperty(JAXP_SCHEMA_SOURCE, new File(schemaSource));
}

// ...
```

In the code above, the variable `schemaSource` relates to a schema source file that you can point the `SAXLocalNameCount` application to by starting it with the `-xsdss` option and providing the name of the schema source file to be used.

Error Handling in the Validating Parser

It is important to recognize that the only reason an exception is thrown when a file fails validation is as a result of the error-handling code shown in Setting up Error Handling. That code is reproduced here as a reminder:

```
// ...

public void warning(SAXParseException spe) throws SAXException {
    out.println("Warning: " + getParseExceptionInfo(spe));
}

public void error(SAXParseException spe) throws SAXException {
    String message = "Error: " + getParseExceptionInfo(spe);
    throw new SAXException(message);
}

public void fatalError(SAXParseException spe) throws SAXException {
    String message = "Fatal Error: " + getParseExceptionInfo(spe);
    throw new SAXException(message);
}

// ...
```

If these exceptions are not thrown, the validation errors are simply ignored. In general, a SAX parsing error is a validation error, although it can also be generated if the file specifies a version of XML that the parser is not prepared to handle. Remember that your application will not generate a validation exception unless you supply an error handler such as the one here.

DTD Warnings

As mentioned earlier, warnings are generated only when the SAX parser is processing a DTD. Some warnings are generated only by the validating parser. The non-validating parser's main goal is to operate as rapidly as possible, but it too generates some warnings.

The XML specification suggests that warnings should be generated as a result of the following:

- Providing additional declarations for entities, attributes, or notations. (Such declarations are ignored. Only the first is used. Also, note that duplicate definitions of elements always

produce a fatal error when validating, as you saw earlier.)

- Referencing an undeclared element type. (A validity error occurs only if the undeclared type is actually used in the XML document. A warning results when the undeclared element is referenced in the DTD.)
- Declaring attributes for undeclared element types.

The Java XML SAX parser also emits warnings in other cases:

- No `<!DOCTYPE ...>` when validating.
- References to an undefined parameter entity when not validating. (When validating, an error results. Although nonvalidating parsers are not required to read parameter entities, the Java XML parser does so. Because it is not a requirement, the Java XML parser generates a warning, rather than an error.)
- Certain cases where the character-encoding declaration does not look right.

Running the SAX Parser Examples with Validation

In this section, the `SAXLocalNameCount` sample program used previously will be used again, except this time it will be run with validation against an XML Schema or a DTD. The best way to demonstrate the different types of validation is to modify the code of the XML file being parsed, as well as the associated schema and DTDs, to break the processing and get the application to generate exceptions.

Experimenting with DTD Validation Errors

As stated above, these examples reuse the `SAXLocalNameCount` program. The locations where you will find the sample and its associated files are given in Running the SAX Parser Example without Validation.

1. **If you have not already done so, navigate to the `samples` directory.** `% cd` *`install-dir`*`/jaxp-1_4_2-`*`release-date`*`/samples`
2. **If you have not already done so, compile the example class.**`% javac sax/*`
3. **Open the file `data/rich_iii.xml` in a text editor.**

   This is the same XML file that was processed without validation in To Run the `SAXLocalNameCount` Example without Validation. At the beginning of `data/rich_iii.xml`, you will see that the `DOCTYPE` declaration points a validating parser to a DTD file called `play.dtd`. If DTD validation is activated, the structure of the XML file being parsed will be checked against the structure provided in `play.dtd`.

4. **Delete the declaration `<!DOCTYPE PLAY SYSTEM "play.dtd">` from the beginning of the file.**

   Do not forget to save the modification, but leave the file open, as it will be needed again later.

5. **Run the `SAXLocalNameCount` program, with DTD validation activated.**

   To do this, you must specify the `-dtd` option when you run the program.

   ```
   % java sax/SAXLocalNameCount -dtd data/rich_iii.xml
   ```

   The result you see will look something like this:

   ```
   Exception in thread "main" org.xml.sax.SAXException:
   ```

```
Error: URI=file:install-dir/JAXP_sources/jaxp-1_4_2-release-date
/samples/data/rich_iii.xml
Line=12: Document is invalid: no grammar found.
```

Note - This message was generated by the JAXP 1.4.2 libraries. If you are using a different parser, the error message is likely to be somewhat different.

This message says that there is no grammar against which the document `rich_iii.txt` can be validated, so therefore it is automatically invalid. In other words, the message is saying that you are trying to validate the document, but no DTD has been declared, because no `DOCTYPE` declaration is present. So now you know that a DTD is a requirement for a valid document. That makes sense.

6. **Restore the `<!DOCTYPE PLAY SYSTEM "play.dtd">` declaration to `data/rich_iii.xml`.**

   Again, do not forget to save the file, but leave it open.

7. **Return to `data/rich_iii.xml` and modify the tags for the character "KING EDWARD The Fourth" in line 26.**

   Change the start and end tags from `<PERSONA>` and `</PERSONA>` to `<PERSON>` and `</PERSON>`. Line 26 should now look like this:

   ```
   26:<PERSON>KING EDWARD The Fourth</PERSON>
   ```

   Again, do not forget to save the modification, and leave the file open.

8. **Run the `SAXLocalNameCount` program, with DTD validation activated.**

   This time, you will see a different error when you run the program:

   ```
   % java sax/SAXLocalNameCount -dtd data/rich_iii.xml
   Exception in thread "main" org.xml.sax.SAXException:
   Error: URI=file:install-dir/JAXP_sources/jaxp-1_4_2-release-date
   /samples/data/rich_iii.xml
   Line=26: Element type "PERSON" must be declared.
   ```

   Here you can see that the parser has objected to an element that is not included in the DTD `data/play.dtd`.

9. **In `data/rich_iii.xml` correct the tags for "KING EDWARD The Fourth".**

   Return the start and end tags to their original versions, `<PERSONA>` and `</PERSONA>`.

10. **In `data/rich_iii.xml`, delete `<TITLE>Dramatis Personae</TITLE>` from line 24.**

    Once more, do not forget to save the modification.

11. **Run the `SAXLocalNameCount` program, with DTD validation activated.**

    As before, you will see another validation error:

    ```
    java sax/SAXLocalNameCount -dtd data/rich_iii.xml
    Exception in thread "main" org.xml.sax.SAXException:
    Error: URI=file:install-dir/JAXP_sources/jaxp-1_4_2-release-date
    /samples/data/rich_iii.xml
    Line=85: The content of element type "PERSONAE" must match "(TITLE,
    (PERSONA|PGROUP)+)".
    ```

By deleting the <TITLE> element from line 24, the <PERSONAE> element is rendered invalid because it does not contain the sub-elements that the DTD expects of a <PERSONAE> element. Note that the error message states that the error is in line 85 of data/rich_iii.xml, even though you deleted the <TITLE> element from line 24. This is because the closing tag of the <PERSONAE> element is located at line 85 and the parser only throws the exception when it reaches the end of the element it parsing.

12. **Open the DTD file, `data/play.dtd` in a text editor.**

    In the DTD file, you can see the declaration of the <PERSONAE> element, as well as all the other elements that can be used in XML documents that conform to the play DTD. The declaration of <PERSONAE> looks like this.

    ```
    <!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
    ```

    As you can see, the <PERSONAE> element requires a <TITLE> sub-element. The pipe (|) key means that either <PERSONA> or <PGROUP> sub-elements can be included in a <PERSONAE> element, and the plus (+) key after the (PERSONA | PGROUP) grouping means that at least one or more of either of these sub-elements must be included.

13. **Add a question mark (?) key after `TITLE` in the declaration of `<PERSONAE>`.**

    Adding a question mark to a sub-element's declaration in a DTD makes the presence of one instance of that sub-element optional.

    ```
    <!ELEMENT PERSONAE (TITLE?, (PERSONA | PGROUP)+)>
    ```

    If you were add an asterisk (*) after the element, you could include either zero or multiple instances of that sub-element. However, in this case it does not make sense to have more than one title in a section of a document.

    Do not forget to save the modification you have made to data/play.dtd.

14. **Run the `SAXLocalNameCount` program, with DTD validation activated.**`% java sax/SAXLocalNameCount -dtd data/rich_iii.xml`

    This time, you should see the proper output of SAXLocalNameCount, with no errors.

Experimenting with Schema Validation Errors

The previous exercise demonstrated using SAXLocalNameCount to validate an XML file against a DTD. In this exercise you will use SAXLocalNameCount to validate a different XML file against both the standard XML Schema definition and a custom schema source file. Again, this type of validation will be demonstrated by breaking the parsing process by modifying the XML file and the schema, so that the parser throws errors.

As stated above, these examples reuse the SAXLocalNameCount program. The locations where you will find the sample and its associated files are given in Running the SAX Parser Example without Validation.

1. **If you have not already done so, navigate to the `samples` directory.**`% cd install-dir/jaxp-1_4_2-release-date/samples`
2. **If you have not already done so, compile the example class.**`% javac sax/*`
3. **Open the file `data/personal-schema.xml` in a text editor.**

   This is a simple XML file that provides the names and contact details for the employees of a small company. In this XML file, you will see that it has been associated with a schema

definition file `personal.xsd`.

```
<personnel xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation='personal.xsd'>
```

4. **Open the file `data/personal.xsd` in a text editor.**

   This schema defines what kinds of information are required about each employee in order for an XML document associated with the schema to be considered valid. For example, by examining the schema definition, you can see that each `person` element requires a `name`, and that each person's name must comprise a `family` name and a `given` name. Employees can also optionally have email addresses and URLs.

5. **In `data/personal.xsd`, change the minimum number of email addresses required for a `person` element from `0` to `1`.**

   The declaration of the `email` element is now as follows.

   ```
   <xs:element ref="email" minOccurs='1' maxOccurs='unbounded'/>
   ```

6. **In `data/personal-schema.xml`, delete the `email` element from the `person` element `one.worker`.**

   Worker One now looks like this:

   ```
   <person id="one.worker">
     <name><family>Worker</family> <given>One</given></name>
     <link manager="Big.Boss"/>
   </person>
   ```

7. **Run `SAXLocalNameCount` against `personal-schema.xml`, with no schema validation.**`% java sax/SAXLocalNameCount data/personal-schema.xml`

   `SAXLocalNameCount` informs you of the number of times each element occurs in `personal-schema.xml`.

   ```
   Local Name "email" occurs 5 times
   Local Name "name" occurs 6 times
   Local Name "person" occurs 6 times
   Local Name "family" occurs 6 times
   Local Name "link" occurs 6 times
   Local Name "personnel" occurs 1 times
   Local Name "given" occurs 6 times
   ```

   You see that `email` only occurs five times, whereas there are six `person` elements in `personal-schema.xml`. So, because we set the minimum occurrences of the `email` element to 1 per `person` element, we know that this document is invalid. However, because `SAXLocalNameCount` was not told to validate against a schema, no error is reported.

8. **Run `SAXLocalNameCount` again, this time specifying that the `personal-schema.xml` document should be validated against a the `personal.xsd` schema definition.**

   As you saw in Validating with XML Schema above, `SAXLocalNameCount` has an option to enable schema validation. Run `SAXLocalNameCount` with the following command.

   ```
   % java sax/SAXLocalNameCount -xsd data/personal-schema.xml
   ```

   This time, you will see the following error message.

   ```
   Exception in thread "main" org.xml.sax.SAXException: Error:
   URI=file:install_dir/samples/data/personal-schema.xml
   ```

```
Line=19: cvc-complex-type.2.4.a: Invalid content was found starting with
element 'link'.
One of '{email}' is expected.
```

9. **Restore the `email` element to the `person` element `one.worker`.**
10. **Run `SAXLocalNameCount` a third time, again specifying that the `personal-schema.xml` document should be validated against a the `personal.xsd` schema definition.**`% java sax/SAXLocalNameCount -xsd data/personal-schema.xml`

    This time you will see the correct output, with no errors.

11. **Open `personal-schema.xml` in a text editor again.**
12. **Delete the declaration of the schema definition `personal.xsd` from the `personnel` element.**

    Remove the italicized code from the `personnel` element.

    `<personnel` *`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation='personal.xsd'/>`*
13. **Run `SAXLocalNameCount`, again specifying schema validation.**`% java sax/SAXLocalNameCount -xsd data/personal-schema.xml`

    Obviously, this will not work, as the schema definition against which to validate the XML file has not been declared. You will see the following error.

    ```
    Exception in thread "main" org.xml.sax.SAXException:
    Error: URI=file:install_dir/samples/data/personal-schema.xml
    Line=8: cvc-elt.1: Cannot find the declaration of element 'personnel'.
    ```

14. **Run `SAXLocalNameCount` again, this time passing it the schema definition file at the command line.**`% java sax/SAXLocalNameCount -xsdss data/personal.xsd data/personal-schema.xml`

    This time you use the `SAXLocalNameCount` option that allows you to specify a schema definition that is not hard-coded into the application. You should see the correct output.

Handling Lexical Events

At this point, you have digested many XML concepts, including DTDs and external entities. You have also learned your way around the SAX parser. The remainder of this lesson covers advanced topics that you will need to understand only if you are writing SAX-based applications. If your primary goal is to write DOM-based applications, you can skip ahead to Document Object Model.

You saw earlier that if you are writing text out as XML, you need to know whether you are in a CDATA section. If you are, then angle brackets (<) and ampersands (&) should be output unchanged. But if you are not in a CDATA section, they should be replaced by the predefined entities `<` and `&`. But how do you know whether you are processing a CDATA section?

Then again, if you are filtering XML in some way, you want to pass comments along. Normally the parser ignores comments. How can you get comments so that you can echo them?

This section answers those questions. It shows you how to use `org.xml.sax.ext.LexicalHandler` to identify comments, CDATA sections, and references to parsed entities.

Comments, CDATA tags, and references to parsed entities constitute lexical information-that is, information that concerns the text of the XML itself, rather than the XML's information content. Most applications, of course, are concerned only with the content of an XML document. Such applications will not use the `LexicalEventListener` API. But applications that output XML text will find it invaluable.

Note - Lexical event handling is an optional parser feature. Parser implementations are not required to support it. (The reference implementation does so.) This discussion assumes that your parser does so.

How the `LexicalHandler` Works

To be informed when the SAX parser sees lexical information, you configure the `XmlReader` that underlies the parser with a `LexicalHandler`. The `LexicalHandler` interface defines the following event-handling methods.

`comment(String comment)`

> Passes comments to the application.

`startCDATA(),endCDATA()`

> Tells when a CDATA section is starting and ending, which tells your application what kind of characters to expect the next time `characters()` is called.

`startEntity(String name),endEntity(String name)`

> Gives the name of a parsed entity.

`startDTD(String name, String publicId, String systemId),endDTD()`

Tells when a DTD is being processed, and identifies it.

To activate the Lexical Handler, your application must extend `DefaultHandler` and implement the `LexicalHandler` interface. Then, you must configure your `XMLReader` instance that the parser delegates to, and configure it to send lexical events to your lexical handler, as shown below.

```
// ...

SAXParser saxParser = factory.newSAXParser();
XMLReader xmlReader = saxParser.getXMLReader();
xmlReader.setProperty("http://xml.org/sax/properties/lexical-handler",
                      handler);
// ...
```

Here, you configure the `XMLReader` using the `setProperty()` method defined in the `XMLReader` class. The property name, defined as part of the SAX standard, is the URN, `http://xml.org/sax/properties/lexical-handler`.

Finally, add something like the following code to define the appropriate methods that will implement the interface.

```
// ...

public void warning(SAXParseException err) {
    // ...
}

public void comment(char[] ch, int start, int length) throws SAXException {
    // ...
}

public void startCDATA() throws SAXException {
    // ...
}

pubic void endCDATA() throws SAXException {
    // ...
}

public void startEntity(String name) throws SAXException {
    // ...
}

public void endEntity(String name) throws SAXException {
    // ...
}

public void startDTD(String name, String publicId, String systemId)
    throws SAXException {
    // ...
}

public void endDTD() throws SAXException {
    // ...
}

private void echoText() {
    // ...
}

// ...
```

This code will transform your parsing application into a lexical handler. All that remains to be done is to give each of these new methods an action to perform.

Using the `DTDHandler` and `EntityResolver`

This section presents the two remaining SAX event handlers: `DTDHandler` and `EntityResolver`. The `DTDHandler` is invoked when the DTD encounters an unparsed entity or a notation declaration. The `EntityResolver` comes into play when a URN (public ID) must be resolved to a URL (system ID).

The `DTDHandler` API

Choosing the Parser Implementation showed a method for referencing a file that contains binary data, such as an image file, using MIME data types. That is the simplest, most extensible mechanism. For compatibility with older SGML-style data, though, it is also possible to define an unparsed entity.

The `NDATA` keyword defines an unparsed entity:

```
<!ENTITY myEntity SYSTEM "..URL.." NDATA gif>
```

The `NDATA` keyword says that the data in this entity is not parseable XML data but instead is data that uses some other notation. In this case, the notation is named `gif`. The DTD must then include a declaration for that notation, which would look something like the following.

```
<!NOTATION gif SYSTEM "..URL..">
```

When the parser sees an unparsed entity or a notation declaration, it does nothing with the information except to pass it along to the application using the `DTDHandler` interface. That interface defines two methods.

- `notationDecl(String name, String publicId, String systemId)`
- `unparsedEntityDecl(String name, String publicId, String systemId, String notationName`

The `notationDecl` method is passed the name of the notation and either the public or the system identifier, or both, depending on which is declared in the DTD. The `unparsedEntityDecl` method is passed the name of the entity, the appropriate identifiers, and the name of the notation it uses.

Note - The `DTDHandler` interface is implemented by the `DefaultHandler` class.

Notations can also be used in attribute declarations. For example, the following declaration requires notations for the GIF and PNG image-file formats.

```
<!ENTITY image EMPTY>
<!ATTLIST image ...  type  NOTATION  (gif | png) "gif">
```

Here, the type is declared as being either gif or png. The default, if neither is specified, is gif.

Whether the notation reference is used to describe an unparsed entity or an attribute, it is up to the application to do the appropriate processing. The parser knows nothing at all about the semantics of the notations. It only passes on the declarations.

The `EntityResolver` API

The `EntityResolver` API lets you convert a public ID (URN) into a system ID (URL). Your application may need to do that, for example, to convert something like `href="urn:/someName"` into `"http://someURL"`.

The `EntityResolver` interface defines a single method:

```
resolveEntity(String publicId, String systemId)
```

This method returns an `InputSource` object, which can be used to access the entity's contents. Converting a URL into an `InputSource` is easy enough. But the URL that is passed as the system ID will be the location of the original document which is, as likely as not, somewhere out on the web. To access a local copy, if there is one, you must maintain a catalog somewhere on the system that maps names (public IDs) into local URLs.

Further Information

The following links provide further useful information about the technologies presented in this lesson.

- For further information on the SAX standard, see the **SAX standard page**:

  http://www.saxproject.org.

- For more information on the StAX pull parser, see:

  The Java Community Process page:

  http://jcp.org/en/jsr/detail?id=173.

  Elliot Rusty Harold's introduction:

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

- For more information on schema-based validation mechanisms, see

  The W3C standard validation mechanism, XML Schema:

  http://www.w3c.org/XML/Schema.

  RELAX NG's regular-expression-based validation mechanism:

  Oasis Relax NG TC.

  Schematron's assertion-based validation mechanism:

  http://www.ascc.net/xml/resource/schematron/schematron.html.