Documentation

The Java™ Tutorials

Trail: Java API for XML Processing (JAXP)
Lesson: Simple API for XML

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 `SAXException`s. 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.

Previous page: When to Use SAX
Next page: Implementing SAX Validation