Echoing an XML File with the SAX Parser

In real life, you will have little need to echo an XML file with a SAX parser. Usually, you'll want to process the data in some way in order to do something useful with it. (If you want to echo it, it's easier to build a DOM tree and use that for output.) But echoing an XML structure is a great way to see the SAX parser in action, and it can be useful for debugging.

In this exercise, you'll echo SAX parser events to `System.out`. Consider it the "Hello World" version of an XML-processing program. It shows you how to use the SAX parser to get at the data and then echoes it to show you what you have.

Note: The code discussed in this section is in `Echo01.java`. The file it operates on is `slideSample01.xml`, as described in Writing a Simple XML File. (The browsable version is `slideSample01-xml.html`.)

Creating the Skeleton

Start by creating a file named `Echo.java` and enter the skeleton for the application:

```
public class Echo
{
  public static void main(String argv[])
  {
    }
}
```

Because you'll run it standalone, you need a `main` method. And you need command-line arguments so that you can tell the application which file to echo.

Importing Classes

Next, add the `import` statements for the classes the application will use:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
public class Echo
{
  ...
```

The classes in `java.io`, of course, are needed to do output. The `org.xml.sax` package defines all the interfaces we use for the SAX parser. The `SAXParserFactory` class creates the instance we use. It throws a `ParserConfigurationException` if it cannot produce a parser that matches the specified configuration of options. (Later, you'll see more about the configuration options.) The `SAXParser` is what the factory returns for parsing, and the `DefaultHandler` defines the class that will handle the SAX events that the parser generates.

Setting Up for I/O

The first order of business is to process the command-line argument, get the name of the file to echo, and set up the output stream. Add the following highlighted text to take care of those tasks and do a bit of additional housekeeping:

```
public static void main(String argv[])

{
  if (argv.length != 1) {
    System.err.println("Usage: cmd filename");
    System.exit(1);
  }
  try {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");
  }
  catch (Throwable t) {
      t.printStackTrace();
  }
  System.exit(0);
}

static private Writer out;
```

When we create the output stream writer, we are selecting the UTF-8 character encoding. We could also have chosen US-ASCII or UTF-16, which the Java platform also supports. For more information on these character sets, see Java Encoding Schemes.

Implementing the ContentHandler Interface

The most important interface for our current purposes 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`, `endElement`, and `characters`.

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. Enter the following highlighted code to extend that class:

```
public class Echo extends DefaultHandler
{
  ...
}
```

Note: `DefaultHandler` also defines do-nothing methods for the other major events, defined in the `DTDHandler`, `EntityResolver`, and `ErrorHandler` interfaces. You'll learn more about those methods as we go along.

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. In the current program, this sequence means that it winds up back at the `Throwable` exception handler at the bottom of the `main` method.

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.

Setting up the Parser

Now (at last) you're ready to set up the parser. Add the following highlighted code to set it up and get it started:

```
public static void main(String argv[])
{
  if (argv.length != 1) {
    System.err.println("Usage: cmd filename");
    System.exit(1);
  }
  // Use an instance of ourselves as the SAX event handler
  DefaultHandler handler = new Echo();
  // Use the default (non-validating) parser
  SAXParserFactory factory = SAXParserFactory.newInstance();
  try {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");
    // Parse the input
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse( new File(argv[0]), handler );
  } catch (Throwable t) {
    t.printStackTrace();
  }
  System.exit(0);
}
```

With these lines of code, you create a `SAXParserFactory` instance, as determined by the setting of the `javax.xml.parsers.SAXParserFactory` system property. You then get a parser from the factory and give the parser an instance of this class to handle the parsing events, telling it which input file to process.

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 `SAXParser`'s `getParser()` method.

For now, you are simply catching any exception that the parser might throw. You'll learn more about error processing in a later section of this chapter, Handling Errors with the Nonvalidating Parser.

Writing the Output

The `ContentHandler` methods throw `SAXExceptions` but not `IOExceptions`, which can occur while writing. The `SAXException` can wrap another exception, though, so it makes sense to do the output in a method that takes care of the exception-handling details. Add the following highlighted code to define an `emit` method that does that:

```
static private Writer out;
private void emit(String s)
throws SAXException
{
  try {
    out.write(s);
    out.flush();
  } catch (IOException e) {
    throw new SAXException("I/O error", e);
  }
}
...
```

When `emit` is called, any I/O error is wrapped in `SAXException` along with a message that identifies it. That exception is then thrown back to the SAX parser. You'll learn more about SAX exceptions later. For

now, keep in mind that `emit` is a small method that handles the string output. (You'll see it called often in later code.)

Spacing the Output

Here is another bit of infrastructure we need before doing some real processing. Add the following highlighted code to define an `nl()` method that writes the kind of line-ending character used by the current system:

```
private void emit(String s)
  ...
}
private void nl()
throws SAXException
{
  String lineEnd = System.getProperty("line.separator");
  try {
    out.write(lineEnd);
  } catch (IOException e) {
    throw new SAXException("I/O error", e);
  }
}
```

Note: Although it seems like a bit of a nuisance, you will be invoking `nl()` many times in later code. Defining it now will simplify the code later on. It also provides a place to indent the output when we get to that section of the tutorial.

Handling Content Events

Finally, let's write some code that actually processes the `ContentHandler` events.

Document Events

Add the following highlighted code to handle the start-document and end-document events:

```
static private Writer out;

public void startDocument()
throws SAXException
{
  emit("<?xml version='1.0' encoding='UTF-8'?>");
  nl();
}

public void endDocument()
throws SAXException
{
  try {
    nl();
    out.flush();
  } catch (IOException e) {
    throw new SAXException("I/O error", e);
  }
}
private void echoText()
...
```

Here, you are echoing an XML declaration when the parser encounters the start of the document. Because you set up `OutputStreamWriter` using UTF-8 encoding, you include that specification as part of the declaration.

Note: However, the IO classes don't understand the hyphenated encoding names, so you specified `UTF8` for the `OutputStreamWriter` rather than `UTF-8`.

At the end of the document, you simply put out a final newline and flush the output stream. Not much going on there.

Element Events

Now for the interesting stuff. Add the following highlighted code to process the start-element and end-element events:

```
public void startElement(String namespaceURI,
        String sName, // simple name
        String qName, // qualified name
        Attributes attrs)
throws SAXException
{
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespace-aware
  emit("<"+eName);
  if (attrs != null) {
    for (int i = 0; i < attrs.getLength(); i++) {
      String aName = attrs.getLocalName(i); // Attr name
      if ("".equals(aName)) aName = attrs.getQName(i);
      emit(" ");
      emit(aName+"=\""+attrs.getValue(i)+"\"");
    }
  }
  emit(">");
}
public void endElement(String namespaceURI,
        String sName, // simple name
        String qName  // qualified name
        )
throws SAXException
{
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespace-aware
  emit("</"+eName+">");
}
private void emit(String s)
...
```

With this code, you echo the element tags, including any attributes defined in the start tag. Note that when the `startElement()` method is invoked, if namespace processing is not enabled, then the simple name (*local name*) for elements and attributes could turn out to be the empty string. The code handles that case by using the qualified name whenever the simple name is the empty string.

Character Events

To finish handling the content events, you need to handle the characters that the parser delivers to your application.

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 accumulate the characters in a buffer and operate on them only when you are sure that all of them have been found.

Add the following highlighted line to define the text buffer:

```
public class Echo01 extends DefaultHandler
{
  StringBuffer textBuffer;

  public static void main(String argv[])
  {
...
```

Then add the following highlighted code to accumulate the characters the parser delivers in the buffer:

```
public void endElement(...)
throws SAXException
{
  ...
}
public void characters(char buf[], int offset, int len)
throws SAXException
{
  String s = new String(buf, offset, len);
  if (textBuffer == null) {
    textBuffer = new StringBuffer(s);
  } else {
    textBuffer.append(s);
  }
}
private void emit(String s)
...
```

Next, add the following highlighted method to send the contents of the buffer to the output stream.

```
public void characters(char buf[], int offset, int len)
throws SAXException
{
  ...
}
private void echoText()
throws SAXException
{
  if (textBuffer == null) return;
  String s = ""+textBuffer;
  emit(s);
  textBuffer = null;
}
private void emit(String s)
...
```

When this method is called twice in a row (which will happen at times, as you'll see next), the buffer will be null. In that case, the method simply returns. When the buffer is not null, however, its contents are sent to the output stream.

Finally, add the following highlighted code to echo the contents of the buffer whenever an element starts or ends:

```
public void startElement(...)
throws SAXException
{
  echoText();
  String eName = sName; // element name
  ...
}
public void endElement(...)
```

```
throws SAXException
{
  echoText();
  String eName = sName; // element name
  ...
}
```

You're finished accumulating text when an element ends, of course. So you echo it at that point, and that action clears the buffer before the next element starts.

But you also want to echo the accumulated text when an element starts! That's 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>`.

Note: Most of the time, though, the accumulated text will be echoed when an `endElement()` event occurs. When a `startElement()` event occurs after that, the buffer will be empty. The first line in the `echoText()` method checks for that case, and simply returns.

Congratulations! At this point you have written a complete SAX parser application. The next step is to compile and run it.

Note: To be strictly accurate, the character handler should scan the buffer for ampersand characters (`&`) and left-angle bracket characters (<) and replace them with the strings `&amp;` or `&lt;`, as appropriate. You'll find out more about that kind of processing when we discuss entity references in Displaying Special Characters and CDATA.

Compiling and Running the Program

In the Application Server, the JAXP libraries are in the directory *<J2EE_HOME>*`/lib/endorsed`. These are newer versions of the standard JAXP libraries than those that are part of the Java 2 platform, Standard Edition versions 1.4.x.

The Application Server automatically uses the newer libraries when a program runs. So you don't have to be concerned with where they reside when you deploy an application. And because the JAXP APIs are identical in both versions, you don't need to be concerned at compile time either. So compiling the program you created is as simple as issuing this command:

```
javac Echo.java
```

But to run the program outside the server container, you must be sure that the `java` runtime finds the newer versions of the JAXP libraries. That situation can occur, for example, when you're unit-testing parts of your application outside of server, as well as here, when you're running the XML tutorial examples.

There are two ways to make sure that the program uses the latest version of the JAXP libraries:

- Copy the *<J2EE_HOME>*`/lib/endorsed` directory to *<J2EE_HOME>*`/jdk/jre/lib/endorsed` (if you are using the Java 2 SDK that comes with the Application Server) or *<JAVA_HOME>*`/jre/lib/endorsed` (if you are using a version of the Java 2 SDK that you have installed separately) You can then run the program with this command:

  <J2SE SDK installation>`/bin/java Echo slideSample.xml`

  The libraries will then be found in the endorsed standards directory.

- Use the endorsed directories system property to specify the location of the libraries, by specifying this option on the `java` command line:

  ```
  -D"java.endorsed.dirs=<J2EE_HOME>/lib/endorsed"
  ```
  or
  ```
  -D"java.endorsed.dirs=<JAVA_HOME>/jre/lib/endorsed
  ```

Note: Because the JAXP *APIs* are already built into the Java 2 platform, Standard Edition, they don't need to be specified at compile time. However, when the JAXP factories instantiate an *implementation*, the endorsed directories mechanism is employed to make sure that the desired implementation is instantiated.

Checking the Output

Here is part of the program's output, showing some of its weird spacing:

```
...
<slideshow title="Sample Slide Show" date="Date of publication"
author="Yours Truly">


  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>
  ...
```

Note: The program's output is contained in `Echo01-01.txt`. (The browsable version is `Echo01-01.html`.)

When we look at this output, a number of questions arise. Where is the excess vertical whitespace coming from? And why are the elements indented properly, when the code isn't doing it? We'll answer those questions in a moment. First, though, there are a few points to note about the output:

- The comment defined at the top of the file

  ```
  <!-- A SAMPLE set of slides -->
  ```

  does not appear in the listing. Comments are ignored unless you implement a `LexicalHandler`. You'll see more on that subject later in this tutorial.
- Element attributes are listed all together on a single line. If your window isn't really wide, you won't see them all.
- The single-tag empty element you defined (`<item/>`) is treated exactly the same as a two-tag empty element (`<item></item>`). It is, for all intents and purposes, identical. (It's just easier to type and consumes less space.)

Identifying the Events

This version of the echo program might be useful for displaying an XML file, but it doesn't tell you much about what's going on in the parser. The next step is to modify the program so that you see where the spaces and vertical lines are coming from.

Note: The code discussed in this section is in `Echo02.java`. The output it produces is shown in `Echo02-01.txt`. (The browsable version is `Echo02-01.html`.)

Make the following highlighted changes to identify the events as they occur:

```
public void startDocument()
throws SAXException
```

```
{
  nl();
  nl();
  emit("START DOCUMENT");
  nl();
  emit("<?xml version='1.0' encoding='UTF-8'?>");
  nl();
}

public void endDocument()
throws SAXException
{
  nl();
  emit("END DOCUMENT");
  try {
  ...
}

public void startElement(...)
throws SAXException
{
  echoText();
  nl();
  emit("ELEMENT: ");
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespac-aware
  emit("<"+eName);
  if (attrs != null) {
    for (int i = 0; i < attrs.getLength(); i++) {
      String aName = attrs.getLocalName(i); // Attr name
      if ("".equals(aName)) aName = attrs.getQName(i);
      emit(" ");
      emit(aName+"=\""+attrs.getValue(i)+"\"");
      nl();
      emit("   ATTR: ");
      emit(aName);
      emit("\t\"");
      emit(attrs.getValue(i));
      emit("\"");
    }
  }
  if (attrs.getLength() > 0) nl();
  emit(">");
}
public void endElement(...)
throws SAXException
{
  echoText();
  nl();
  emit("END_ELM: ");
  String eName = sName; // element name
  if ("".equals(eName)) eName = qName; // not namespace-aware
  emit("<"+eName+">");
}

...
private void echoText()
throws SAXException
{
  if (textBuffer == null) return;
  nl();
```

```
    emit("CHARS: |");
    String s = ""+textBuffer;
    emit(s);
    emit("|");
    textBuffer = null;
  }
```

Compile and run this version of the program to produce a more informative output listing. The attributes are now shown one per line, and that is nice. But, more importantly, output lines such as the following show that both the indentation space and the newlines that separate the attributes come from the data that the parser passes to the `characters()` method.

```
    CHARS: |

  |
```

Note: The XML specification requires all input line separators to be normalized to a single newline. The newline character is specified as in Java, C, and UNIX systems, but goes by the alias "linefeed" in Windows systems.

Compressing the Output

To make the output more readable, modify the program so that it outputs only characters whose values are something other than whitespace.

Note: The code discussed in this section is in `Echo03.java`.

Make the following changes to suppress output of characters that are all whitespace:

```
public void echoText()
throws SAXException
{
  nl();
  ~~emit("CHARS: |");~~
  emit("CHARS:   ");
  String s = ""+textBuffer;
  if (!s.trim().equals("")) emit(s);
  ~~emit("|");~~
}
```

Next, add the following highlighted code to echo each set of characters delivered by the parser:

```
public void characters(char buf[], int offset, int len)
throws SAXException
{
  if (textBuffer != null) {
    echoText();
    textBuffer = null;
  }
  String s = new String(buf, offset, len);
  ...
}
```

If you run the program now, you will see that you have also eliminated the indentation, because the indent space is part of the whitespace that precedes the start of an element. Add the following highlighted code to manage the indentation:

```
static private Writer out;
private String indentString = "    "; // Amount to indent
```

```
private int indentLevel = 0;

...
public void startElement(...)
throws SAXException
{
  indentLevel++;
  nl();
  emit("ELEMENT: ");
  ...
}
public void endElement(...)
throws SAXException
{
  nl();
  emit("END_ELM: ");
  emit("</"+sName+">");
  indentLevel--;
}
...
private void nl()
throws SAXException
{
  ...
  try {
    out.write(lineEnd);
    for (int i=0; i < indentLevel; i++)
      out.write(indentString);
  } catch (IOException e) {
  ...
}
```

This code sets up an indent string, keeps track of the current indent level, and outputs the indent string whenever the `nl` method is called. If you set the indent string to "", the output will not be indented. (Try it. You'll see why it's worth the work to add the indentation.)

You'll be happy to know that you have reached the end of the "mechanical" code in the Echo program. From this point on, you'll be doing things that give you more insight into how the parser works. The steps you've taken so far, though, have given you a lot of insight into how the parser sees the XML data it processes. You have also gained a helpful debugging tool that you can use to see what the parser sees.

Inspecting the Output

Here is part of the output from this version of the program:

```
ELEMENT: <slideshow
...
>
CHARS:

CHARS:

  ELEMENT: <slide
  ...
  END_ELM: </slide>
CHARS:

CHARS:
```

Note: The complete output is `Echo03-01.txt`. (The browsable version is `Echo03-01.html`.)

Note that the `characters` method is invoked twice in a row. Inspecting the source file `slideSample01.xml` shows that there is a comment before the first slide. The first call to `characters`

comes before that comment. The second call comes after. (Later, you'll see how to be notified when the parser encounters a comment, although in most cases you won't need such notifications.)

Note, too, that the `characters` method is invoked after the first slide element, as well as before. When you are thinking in terms of hierarchically structured data, that seems odd. After all, you intended for the `slideshow` element to contain `slide` elements and not text. Later, you'll see how to restrict the `slideshow` element by using a DTD. When you do that, the `characters` method will no longer be invoked.

In the absence of a DTD, though, the parser must assume that any element it sees contains text such as that in the first item element of the overview slide:

```
<item>Why <em>WonderWidgets</em> are great</item>
```

Here, the hierarchical structure looks like this:

```
ELEMENT:        <item>
CHARS:        Why
  ELEMENT:        <em>
  CHARS:         WonderWidgets
  END_ELM:        </em>
CHARS:         are great
END_ELM:        </item>
```

Documents and Data

In this example, it's clear that there are characters intermixed with the hierarchical structure of the elements. The fact that text can surround elements (or be prevented from doing so with a DTD or schema) helps to explain why you sometimes hear talk about "XML data" and other times hear about "XML documents." XML comfortably handles both structured data and text documents that include markup. The only difference between the two is whether or not text is allowed between the elements.

Note: In a later section of this tutorial, you will work with the `ignorableWhitespace` method in the `ContentHandler` interface. This method can be invoked only when a DTD is present. If a DTD specifies that `slideshow` does not contain text, then all the whitespace surrounding the `slide` elements is by definition ignorable. On the other hand, if `slideshow` can contain text (which must be assumed to be true in the absence of a DTD), then the parser must assume that spaces and lines it sees between the `slide` elements are significant parts of the document.

All of the material in *The J2EE(TM) 1.4 Tutorial* is copyright-protected and may not be published in other works without express written permission from Sun Microsystems.