Quickstart

This document provides a quick-start tutorial for Java programmers who wish to use SAX2 in their programs.

Requirements

SAX is a common interface implemented for many different XML parsers (and things that pose as XML parsers), just as the JDBC is a common interface implemented for many different relational databases (and things that pose as relational databases). If you want to use SAX, you'll need all of the following:

- Java 1.1 or higher.
- A SAX2-compatible XML parser installed on your Java classpath. (If you need such a parser, see the page of links at the left.)
- The SAX2 distribution installed on your Java classpath. (This probably came with your parser.)

Most Java/XML tools distributions include SAX2 and a parser using it. Most web applications servers use it for their core XML support. In particular, environments with JAXP 1.1 support include SAX2.

Parsing a document

Start by creating a class that extends DefaultHandler:

```
import org.xml.sax.helpers.DefaultHandler;

public class MySAXApp extends DefaultHandler
{

    public MySAXApp ()
    {
        super();
    }

}
```

Since this is a Java application, we'll create a static *main* method that uses the the createXMLReader method from the XMLReaderFactory class to choose a SAX driver dynamically. Note the "throws Exception" wimp-out; real applications would need real error handling:

```
    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
    }
```

In case your Java environment did not arrange for a compiled-in default (or to use the `META-INF/services/org.xl.sax.driver` system resource), you'll probably need to set the `org.xml.sax.driver` Java system property to the full classname of the SAX driver, as in

```
java -Dorg.xml.sax.driver=com.example.xml.SAXDriver MySAXApp sample.xml
```

Several of the SAX2 drivers currently in in widespread use are listed on the "links" page. Class names you might use include:

Class Name	Notes
`gnu.xml.aelfred2.SAXDriver`	Lightweight non-validating parser; Free Software
`gnu.xml.aelfred2.XmlReader`	Optionally validates; Free Software
`oracle.xml.parser.v2.SAXParser`	Optionally validates; proprietary
`org.apache.crimson.parser.XMLReaderImpl`	Optionally validates; used in JDK 1.4; Open Source
`org.apache.xerces.parsers.SAXParser`	Optionally validates; Open Source

Alternatively, if you don't mind coupling your application to a specific SAX driver, you can use its constructor directly. We assume that the SAX driver for your XML parser is named `com.example.xml.SAXDriver`, but this does not really exist. You *must* know the name of the real driver for your parser to use this approach.

```
    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = new com.example.xml.SAXDriver();
    }
```

We can use this object to parse XML documents, but first, we have to register event handlers that the parser can use for reporting information, using the setContentHandler and setErrorHandler methods from the XMLReader

interface. In a real-world application, the handlers will usually be separate objects, but for this simple demo, we've bundled the handlers into the top-level class, so we just have to instantiate the class and register it with the XML reader:

```
    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);
    }
```

This code creates an instance of MySAXApp to receive XML parsing events, and registers it with the XML reader for regular content events and error events (there are other kinds, but they're rarely used). Now, let's assume that all of the command-line args are file names, and we'll try to parse them one-by-one using the parse method from the XMLReader interface:

```
    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

                                // Parse each file provided on the
                                // command line.
        for (int i = 0; i < args.length; i++) {
            FileReader r = new FileReader(args[i]);
            xr.parse(new InputSource(r));
        }
    }
```

Note that each reader must be wrapped in an InputSource object to be parsed. Here's the whole demo class together (so far):

```
import java.io.FileReader;

import org.xml.sax.XMLReader;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;


public class MySAXApp extends DefaultHandler
{

    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

                                // Parse each file provided on the
                                // command line.
        for (int i = 0; i < args.length; i++) {
            FileReader r = new FileReader(args[i]);
            xr.parse(new InputSource(r));
        }
    }


    public MySAXApp ()
    {
        super();
    }
}
```

You can compile this code and run it (make sure you specify the SAX driver class in the org.xml.sax.driver property), but nothing much will happen unless the document contains malformed XML, because you have not yet set up your application to handle SAX events.

Handling events

Things get interesting when you start implementing methods to respond to XML parsing events (remember that we registered our class to receive XML parsing events in the previous section). The most important events are the start and end of the document, the start and end of elements, and character data.

To find out about the start and end of the document, the client application implements the startDocument and

endDocument methods:

```
public void startDocument ()
{
    System.out.println("Start document");
}

public void endDocument ()
{
    System.out.println("End document");
}
```

The start/endDocument event handlers take no arguments. When the SAX driver finds the beginning of the document, it will invoke the *startDocument* method once; when it finds the end, it will invoke the *endDocument* method once (even if there have been errors).

These examples simply print a message to standard output, but your application can contain any arbitrary code in these handlers: most commonly, the code will build some kind of an in-memory tree, produce output, populate a database, or extract information from the XML stream.

The SAX driver will signal the start and end of elements in much the same way, except that it will also pass some parameters to the startElement and endElement methods:

```
public void startElement (String uri, String name,
                          String qName, Attributes atts)
{
    if ("".equals (uri))
        System.out.println("Start element: " + qName);
    else
        System.out.println("Start element: {" + uri + "}" + name);
}

public void endElement (String uri, String name, String qName)
{
    if ("".equals (uri))
        System.out.println("End element: " + qName);
    else
        System.out.println("End element:   {" + uri + "}" + name);
}
```

These methods print a message every time an element starts or ends, with any Namespace URI in braces before the element's local name. The *qName* contains the raw XML 1.0 name, which you must use for all elements that don't have a namespace URI. In this quick introduction, we won't look at how attributes are accessed; you can access them by name, or by iterating through them much as if they were a vector.

Finally, SAX2 reports regular character data through the characters method; the following implementation will print all character data to the screen; it is a little longer because it pretty-prints the output by escaping special characters:

```
public void characters (char ch[], int start, int length)
{
    System.out.print("Characters:    \"");
    for (int i = start; i < start + length; i++) {
        switch (ch[i]) {
        case '\\':
            System.out.print("\\\\");
            break;
        case '"':
            System.out.print("\\\"");
            break;
        case '\n':
            System.out.print("\\n");
            break;
        case '\r':
            System.out.print("\\r");
            break;
        case '\t':
            System.out.print("\\t");
            break;
        default:
            System.out.print(ch[i]);
            break;
        }
    }
    System.out.print("\"\n");
}
```

Note that a SAX driver is free to chunk the character data any way it wants, so you cannot count on all of the character data content of an element arriving in a single *characters* event.

Sample SAX2 application

Here is the complete sample application (again, in a serious app the event handlers would probably be implemented in a separate class):

```
import java.io.FileReader;

import org.xml.sax.XMLReader;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;


public class MySAXApp extends DefaultHandler
{

    public static void main (String args[])
        throws Exception
    {
        XMLReader xr = XMLReaderFactory.createXMLReader();
        MySAXApp handler = new MySAXApp();
        xr.setContentHandler(handler);
        xr.setErrorHandler(handler);

                                // Parse each file provided on the
                                // command line.
        for (int i = 0; i < args.length; i++) {
            FileReader r = new FileReader(args[i]);
            xr.parse(new InputSource(r));
        }
    }


    public MySAXApp ()
    {
        super();
    }


    //////////////////////////////////////////////////////////////////
    // Event handlers.
    //////////////////////////////////////////////////////////////////


    public void startDocument ()
    {
        System.out.println("Start document");
    }


    public void endDocument ()
    {
        System.out.println("End document");
    }


    public void startElement (String uri, String name,
                              String qName, Attributes atts)
    {
        if ("".equals (uri))
            System.out.println("Start element: " + qName);
        else
            System.out.println("Start element: {" + uri + "}" + name);
    }


    public void endElement (String uri, String name, String qName)
    {
        if ("".equals (uri))
            System.out.println("End element: " + qName);
        else
            System.out.println("End element:   {" + uri + "}" + name);
    }


    public void characters (char ch[], int start, int length)
    {
        System.out.print("Characters:    \"");
        for (int i = start; i < start + length; i++) {
            switch (ch[i]) {
            case '\\':
                System.out.print("\\\\");
                break;
            case '"':
                System.out.print("\\\"");
                break;
            case '\n':
                System.out.print("\\n");
```

```
                System.out.print("\\r");
                break;
            case '\t':
                System.out.print("\\t");
                break;
            default:
                System.out.print(ch[i]);
                break;
            }
        }
        System.out.print("\"\n");
    }

}
```

Sample Output

Consider the following XML document:

```
<?xml version="1.0"?>

<poem xmlns="http://www.megginson.com/ns/exp/poetry">
<title>Roses are Red</title>
<l>Roses are red,</l>
<l>Violets are blue;</l>
<l>Sugar is sweet,</l>
<l>And I love you.</l>
</poem>
```

If this document is named `roses.xml` and there is a SAX2 driver on your classpath named `com.example.xml.SAXDriver` (this driver does not actually exist), you can invoke the sample application like this:

```
java -Dorg.xml.sax.driver=com.example.xml.SAXDriver MySAXApp roses.xml
```

When you run this, you'll get output something like this:

```
Start document
Start element: {http://www.megginson.com/ns/exp/poetry}poem
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}title
Characters:    "Roses are Red"
End element:   {http://www.megginson.com/ns/exp/poetry}title
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Roses are red,"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Violets are blue;"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "Sugar is sweet,"
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
Start element: {http://www.megginson.com/ns/exp/poetry}l
Characters:    "And I love you."
End element:   {http://www.megginson.com/ns/exp/poetry}l
Characters:    "\n"
End element:   {http://www.megginson.com/ns/exp/poetry}poem
End document
```

Note that even this short document generates (at least) 25 events: one for the start and end of each of the six elements used (or, if you prefer, one for each start tag and one for each end tag), one of each of the eleven chunks of character data (including whitespace between elements), one for the start of the document, and one for the end.

If the input document did not include the `xmlns="http://www.megginson.com/ns/exp/poetry"` attribute to declare that all the elements are in that namespace, the output would instead be like:

```
Start document
Start element: poem
Characters:    "\n"
Start element: title
Characters:    "Roses are Red"
End element:   title
Characters:    "\n"
Start element: l
Characters:    "Roses are red,"
End element:   l
Characters:    "\n"
```

```
Start element: l
Characters:    "Violets are blue;"
End element:   l
Characters:    "\n"
Start element: l
Characters:    "Sugar is sweet,"
End element:   l
Characters:    "\n"
Start element: l
Characters:    "And I love you."
End element:   l
Characters:    "\n"
End element:   poem
End document
```

You will most likely work with both types of documents: ones using XML namespaces, and ones not using them. You may also work with documents that have some elements (and attributes) with namespaces, and some without. Make sure that your code actually tests for namespace URIs, rather than assuming they are always present (or always missing).