Understanding SAX

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial introduction

Should I take this tutorial?

This tutorial examines the use of the Simple API for XML version 2.0, or SAX 2.0. It is aimed at developers who have an understanding of XML and wish to learn this lightweight, event-based API for working with XML data. It assumes that you are familiar with concepts such as well-formedness and the tag-like nature of an XML document. (You can get a basic grounding in XML itself through the *Introduction to XML* tutorial, if necessary.) In this tutorial, you will learn how to use SAX to retrieve, manipulate, and output XML data.

Prerequisites: SAX is available in a number of programming languages, such as Java, Perl, C++, and Python. This tutorial uses Java in its demonstrations, but the concepts are substantially similar in all languages, and you can gain a thorough understanding of SAX without actually working the examples.

What is SAX?

The standard means for reading and manipulating XML files is DOM, the Document Object Model. Unfortunately, this method, which involves reading the entire file and storing it in a tree structure, can be inefficient, slow, and it can be a strain on resources.

One alternative is the Simple API for XML, or SAX. SAX allows you to process a document as it's being read, which avoids the need to wait for all of it to be stored before taking action.

SAX was developed by the members of the XML-DEV mailing list, and the Java version is maintained by David Megginson (see Resources on page 28). Their purpose was to provide a more natural means for working with XML that did not involve the overhead of the DOM.

The result was an API that is *event based*. The parser sends events, such as the start or end of an element, to an event handler, which processes the information. The application itself can then deal with the data. The original document remains untouched, but SAX provides the means for manipulating the data, which can then be directed to another process or document.

There is no official standards body for SAX; it is not maintained by the World Wide Web Consortium (W3C) or any other official body, but it is a de facto standard in the XML community.

Tools

The examples in this tutorial, should you decide to try them out, require the following tools to be installed and working correctly. Running the examples is not a requirement for understanding.

A text editor: XML files are simply text. To create and read them, a text editor is all you need.

JavaTM 2 SDK, Standard Edition version 1.3.1: The sample applications demonstrate manipulation of the DOM through Java. You can download the Java SDK from

http://java.sun.com/j2se/1.3/ .

JavaTM APIs for XML Processing: Also known as JAXP 1.1, this is the reference implementation that Sun provides. You can download JAXP from *http://java.sun.com/xml/xml_jaxp.html* .

Other Languages: Should you wish to adapt the examples, SAX implementations are also available in other programming languages. You can find information on Perl, C++, COM, and Python implementations of a SAX parser at *http://www.megginson.com/SAX/applications.html* .

Conventions used in this tutorial

* Text that needs to be typed is displayed in a **`bold monospace`** font.
* *Emphasis/Italics* is used to draw attention to windows, dialog boxes, and/or feature names.
* A `monospace` font is used for file and path names.

About the author

Nicholas Chase has been involved in Web site development for companies including Lucent Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida, and is the author of three books on Web development, including *Java and XML From Scratch* (Que). He loves to hear from readers and can be reached at *nicholas@nicholaschase.com* .

Section 2. DOM, SAX, and when each is appropriate

How SAX processing works

SAX analyzes an XML stream as it goes by, much like an old tickertape. Consider the following XML code snippet:

```
<?xml version="1.0"?>
<samples>
   <server>UNIX</server>
   <monitor>color</monitor>
</samples>
```

A SAX processor analyzing this code snipped would generate, in general, the following events:

```
Start document
Start element (samples)
Characters (white space)
Start element (server)
Characters (UNIX)
End element (server)
Characters (white space)
Start element (monitor)
Characters (color)
End element (monitor)
Characters (white space)
End element (samples)
```

The SAX API allows a developer to capture these events and act on them.

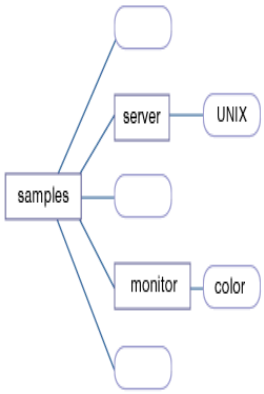
SAX processing involves the following steps:

* Create an event handler.
* Create the SAX parser.
* Assign the event handler to the parser.
* Parse the document, sending each event to the handler.

The pros and cons of event-based processing

The advantages of this kind of processing are much like the advantages of streaming media; analysis can get started immediately, rather than having to wait for all of the data to be processed. Also, because the application is simply examining the data as it goes by, it doesn't need to store it in memory. This is a huge advantage when it comes to large documents. In general, SAX is also much faster than the alternative, the Document Object Model.

On the other hand, because the application is not storing the data in any way, it is impossible to make changes to it using SAX, or to move "backward" in the data stream.



DOM and tree-based processing

The Document Object Model, or DOM, is the "traditional" way of handling XML data. With DOM the data is loaded into memory in a tree-like structure.

For instance, the same document used as an example in the preceding panel would be represented as nodes, as shown to the left.

The rectangular boxes represent element nodes, and the ovals represent text nodes.

DOM uses a root node and parent-child relationships. For instance, in this case, `samples` would be the root node with five children: three text nodes (the white space), and the two element nodes, `server` and `monitor`.

One important thing to realize is that the `server` and `monitor` actually have values of `null`. Instead, they have text nodes for children, `UNIX` and `color`.

Pros and cons of tree-based processing

DOM, and by extension tree-based processing, has several advantages. First, because the tree is persistent in memory, it can be modified so an application can make changes to the data and the structure. It can also work its way up and down the tree at any time, as opposed to the "one-shot deal" of SAX. DOM can also be much simpler to use.

On the other hand, there is a lot of overhead involved in building these trees in memory. It's not unusual for large files to completely overrun a system's capacity. In addition, creating a DOM tree can be a very slow process.

How to choose between SAX and DOM

Whether you choose DOM or SAX is going to depend on several factors:

* Purpose of the application: If you are going to have to make changes to the data and output it as XML, then in most cases, DOM is the way to go. This is particularly true if the changes are to the data itself, as opposed to a simple structural change that can be accomplished with XSL transformations.
* Amount of data: For large files, SAX is a better bet.
* How the data will be used: If only a small amount of the data will actually be used, you may be better off using SAX to extract it into your application. On the other hand, if you

know that you will need to refer back to information that has already been processed, SAX is probably not the right choice.
* The need for speed: SAX implementations are normally faster than DOM implementations.

It's important to remember that SAX and DOM are not mutually exclusive. You can use DOM to create a SAX stream of events, and you can use SAX to create a DOM tree. In fact, most parsers used to create DOM trees are actually using SAX to do it!

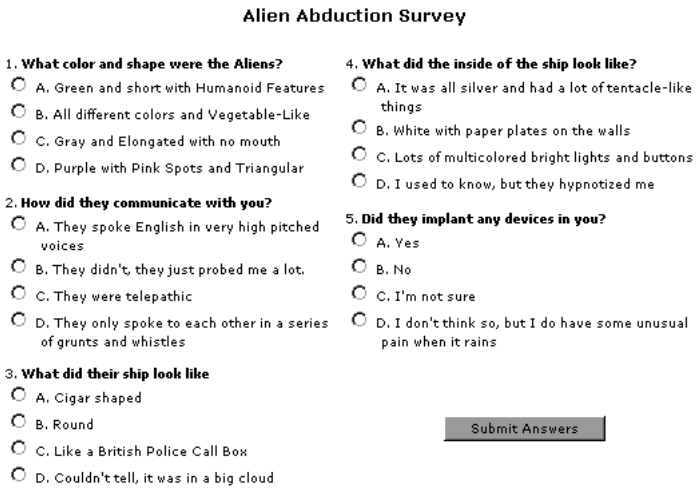
Section 3. Creating the SAX parser

The sample file

This tutorial demonstrates the construction of an application that uses SAX to tally the responses from a group of users asked to take a survey regarding their alien abduction experiences.

The XML code for the survey form, and resultant form are shown below.

```
<?xml version="1.0"?>
<surveys>
<response username="bob">
   <question subject="appearance">A</question>
   <question subject="communication">B</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
   <question subject="implant">B</question>
</response>
<response username="sue">
   <question subject="appearance">C</question>
   <question subject="communication">A</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
   <question subject="implant">A</question>
</response>
<response username="carol">
   <question subject="appearance">A</question>
   <question subject="communication">C</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
   <question subject="implant">C</question>
</response>
</surveys>
```



Creating an event handler

Before an application can use SAX to process an XML document, it must create an event handler. SAX provides a class, `DefaultHandler`, that applications can extend.

Upon the creation of a new instance of the `SurveyReader` class, the constructor

SurveyReader() is executed. This object now has all methods available to it, not just those that are static.

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

 public class SurveyReader extends DefaultHandler
 {

    public SurveyReader() {
       System.out.println("Object Created.");
    }

    public void showEvent(String name) {
       System.out.println("Hello, "+name+"!");
    }

    public static void main (String args[]) {

       SurveyReader reader = new SurveyReader();
       reader.showEvent("Nick");
    }

 }
```

Compiling and running this application displays the "Object Created." message, as well as the greeting.

```
Object Created.
Hello, Nick!
```

Specifying the SAX driver directly

With the event handler in place, the next step is to create the parser, or XMLReader, using the SAX driver. Creating the parser can be accomplished in one of three ways:

* Call the driver directly.
* Allow the driver to be specified at run time.
* Pass the driver as an argument for createXMLReader()

If you know the name of the class that is the SAX driver, you can call it directly. For instance, if the class was (the nonexistent) com.nc.xml.SAXDriver, you could say:

```
 try {
    XMLReader xmlReader = new com.nc.xml.SAXDriver();
 } catch (Exception e) {}
```

This will directly create the XMLReader.

You can make your application more flexible by allowing the class to be specified when the application runs. For example, on the command line you would type:

```
 java -Dorg.xml.sax.driver=com.nc.xml.SAXDriver SurveyReader
```

This makes the information available to the XMLReaderFactory class, so you can say:

```
 try {
    XMLReader xmlReader = XMLReaderFactory.createXMLReader();
 } catch (Exception e) {}
```

If you know the driver name, you can also pass it directly as an argument for `createXMLReader()`.

Using JAXP to create the parser

This example uses a pair of classes that are part of JAXP, `SAXParserFactory` and `SAXParser`, to create the parser without having to know the name of the driver itself.

First you declare the `XMLReader`, `xmlReader`. Then you use `SAXParserFactory` to create a `SAXParser`. It's the `SAXParser` that gives you the `XMLReader`.

```
import org.xml.sax.helpers.DefaultHandler;
 import javax.xml.parsers.SAXParser;
 import javax.xml.parsers.SAXParserFactory;
 import org.xml.sax.XMLReader;

 public class SurveyReader
                     extends DefaultHandler
 {

    public SurveyReader() {
    }

    public static void main (String args[]) {

       XMLReader xmlReader = null;

       try {

          SAXParserFactory spfactory =
              SAXParserFactory.newInstance();

          SAXParser saxParser =
                    spfactory.newSAXParser();

          xmlReader = saxParser.getXMLReader();

       } catch (Exception e) {
             System.err.println(e);
             System.exit(1);
       }
    }
 }
```

Validating vs. non-validating parsers

In order to do anything with an XML document, you have to read the information in it. The application that does this is called a *parser*. The parser reads the data in the file and translates it into the `Document` object.

There are two kinds of parsers: *non-validating* and *validating*.

A non-validating parser is satisfied if the file is well formed. It takes each unit of information and adds it to the document without regard to the actual content.

A validating parser, on the other hand, reads the DOCTYPE declaration and retrieves a file of definitions, called a Document Type Definition, or DTD, for the data. Files can also be described with an XML Schema document, but the syntax is different. In either case, the

parser checks the document to make sure that each element and attribute is defined and contains the proper types of content. For instance, we might specify that every `order` must have a `status`. If we tried to create one without it, a validating parser would signal a problem.

Documents that have been verified by a validating parser are said to be *valid* documents.

Set validation options

Without a specified DTD or schema for the sample document, you can turn validation off by setting a value in the SAX`ParserFactory`. This effects any parsers the factory creates.

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

    XMLReader xmlReader = null;

    try {

       SAXParserFactory spfactory =
            SAXParserFactory.newInstance();
       spfactory.setValidating(false);
       SAXParser saxParser =
                  spfactory.newSAXParser();

       xmlReader = saxParser.getXMLReader();

    } catch (Exception e) {
          System.err.println(e);
          System.exit(1);
    }
 }
 ...
```

Set the content handler

Once you've created the parser, you need to set a `SurveyReader` as the content handler so that it receives the events.

The `setContentHandler()` method of `xmlReader` accomplishes this.

```
...
     xmlReader = saxParser.getXMLReader();
     xmlReader.setContentHandler(new SurveyReader());

  } catch (Exception e) {
 ...
```

This is not, of course, the only option for a content handler. You'll see how to use this method in Serializing a SAX stream on page 24later in this tutorial.

Parse the InputSource

In order to actually parse a file (or anything else, for that matter!) you need an
. This SAX class wraps whatever data you're going to process, so you don't have to worry (too much) about where it's coming from.

Now you're ready to actually parse the file. The application passes the file, wrapped in the `InputSource`, to `parse()` and away it goes.

```
...
 import org.xml.sax.InputSource;
 ...
     xmlReader = saxParser.getXMLReader();
     xmlReader.setContentHandler(new SurveyReader());

     InputSource source =
                     new InputSource("surveys.xml");
     xmlReader.parse(source);

  } catch (Exception e) {
 ...
```

You can compile and run the program, but at this point nothing should happen because the application doesn't have any events defined yet.

Create an ErrorHandler

At this point nothing *should* happen. But of course there's always the chance that there are problems with the data that you're trying to parse. In such a situation, it would be helpful to have a handler for the errors, as well as for the content.

You can create an error handler just as you created a content handler. Normally, you would create this as a separate instance of `ErrorHandler`, but to simplify the example, error handling is included right in `SurveyResults`. This dual usage is possible because the class extends `DefaultHandler` and not `ContentHandler`.

There are three events that you need to be concerned with: warnings, errors, and fatal errors.

```
...
 import org.xml.sax.SAXParseException;

 public class SurveyReader
                         extends DefaultHandler
 {

    public SurveyReader() {
    }

    public void error (SAXParseException e) {
       System.out.println("Error parsing the file: "+e.getMessage());
    }

    public void warning (SAXParseException e) {
       System.out.println("Problem parsing the file: "+e.getMessage());
    }

    public void fatalError (SAXParseException e) {
       System.out.println("Error parsing the file: "+e.getMessage());
       System.out.println("Cannot continue.");
       System.exit(1);
    }

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

Set the ErrorHandler

Once you have the `ErrorHandler` created, setting it up to receive events from the parser is similar to setting up the `ContentHandler`.

Again, this would normally be a separate class, but for now just create a new instance of `SurveyReader`.

```
...
 xmlReader.setContentHandler(new SurveyReader());
 xmlReader.setErrorHandler(new SurveyReader());

 InputSource source = new InputSource("surveys.xml");
 ...
```

Section 4. Event handlers and the SAX events

startDocument()

Now you're set up to parse the document, but nothing actually happens because you have not yet defined any parsing events.

Start by noting the beginning of the document using the `startDocument()` event. This event, like the other SAX events, throws a `SAXException`.

```
...
 import org.xml.sax.SAXException;

 public class SurveyReader
                        extends DefaultHandler
 {
 ...
    public void fatalError (SAXParseException e) {
       System.out.println("Error parsing " +
                     "the file: "+e.getMessage());
       System.out.println("Cannot continue.");
       System.exit(1);
    }

    public void startDocument()
                    throws SAXException {
       System.out.println(
                     "Tallying survey results...");
    }

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

```
Tallying survey results...
```

---

startElement()

Now let's begin looking at the actual data. For each element, the example echoes back the name which is passed to the `startElement()` event (see the Element Listing screenshot below).

SAX version 2.0 added support for Namespaces on page 21, so what the parser actually passes is: the namespace information for this element; the actual name of the element, or `localName`; the combination of the Namespace alias and the `localName` (otherwise known as the qualified name, or `qname`); and any attributes for the element.

```
...
 import org.xml.sax.Attributes;

 public class SurveyReader
                        extends DefaultHandler
 {
 ...
    public void startDocument()
                    throws SAXException {
       System.out.println(
               "Tallying survey results...");
    }

    public void startElement(
                 String namespaceURI,
```

```
                String localName,
                String qName,
                Attributes atts)
                   throws SAXException {

        System.out.print("Start element: ");
        System.out.println(localName);

    }

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

```
Tallying survey results...
Start element: surveys
Start element: response
Start element: question
Start element: question
Start element: question
Start element: question
Start element: question
Start element: response
Start element: question
Start element: question
Start element: question
Start element: question
Start element: question
Start element: response
Start element: question
Start element: question
Start element: question
Start element: question
Start element: question
```

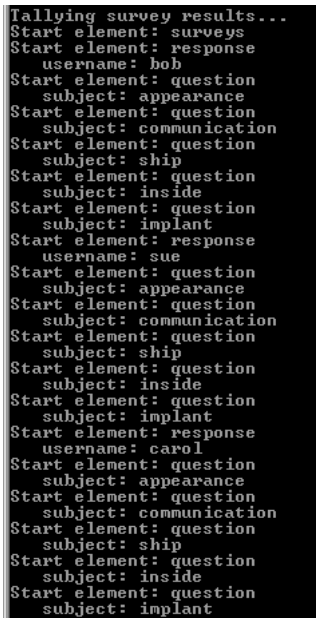
---

startElement(): retrieve attributes

The `startElement()` event also provides access to the attributes for an element. They are passed in within a data structure called Attributes (which replaces the now-deprecated AttributeList).

We can retrieve an attribute value based on its position in the array, or based on the name of the attribute, as we can see to the left.

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

     System.out.print("Start element: ");
     System.out.println(localName);

     for (int att = 0;
          att < atts.getLength();
          att++) {
        String attName = atts.getLocalName(att);
        System.out.println("   "
                     + attName + ": "
                     + atts.getValue(attName));
    }
 }
 ...
```



---

endElement()

One thing that you might notice in the previous example is that the results are not the "pretty printed," or indented, version of XML that we're used to seeing.

There are lots of good reasons to note the end of an element. Start by making the output a little bit easier to decipher by adding indents based on how many levels deep the element is.

Create a simple indent routine, and increase the value of the indent when a new element starts, decreasing it again when the element ends.

```
...
 int indent = 0;
 public void startDocument()
                throws SAXException {
    System.out.println("Tallying survey results...");
    indent = -4;
 }

 public void printIndent(int indentSize) {
    for (int s = 0; s < indentSize; s++) {
       System.out.print(" ");
    }
 }

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

    indent = indent + 4;
    printIndent(indent);

    System.out.print("Start element: ");
    System.out.println(localName);

    for (int att = 0;
         att < atts.getLength();
```

```
        att++) {
      printIndent(indent + 4);
      String attName = atts.getLocalName(att);
      System.out.println("   "
                         + attName + ": "
                         + atts.getValue(attName));
   }
}

public void endElement(String namespaceURI,
                       String localName,
                       String qName)
                           throws SAXException {

   printIndent(indent);
   System.out.println("End Element: "+localName);
   indent = indent - 4;
}
...
```

```
Tallying survey results...
Start element: surveys
Start element: response
   username: bob
Start element: question
   subject: appearance
Start element: question
   subject: communication
Start element: question
   subject: ship
Start element: question
   subject: inside
Start element: question
   subject: implant
Start element: response
   username: sue
Start element: question
   subject: appearance
Start element: question
   subject: communication
Start element: question
   subject: ship
Start element: question
   subject: inside
Start element: question
   subject: implant
Start element: response
   username: carol
Start element: question
   subject: appearance
Start element: question
   subject: communication
Start element: question
   subject: ship
Start element: question
   subject: inside
Start element: question
   subject: implant
```

---

characters()

Now you've got the elements, so go ahead and retrieve the actual data using `characters()`. Take a look at the signature of this method for a moment:

```
public void characters(char[] ch,
                       int start,
                       int length)
                       throws SAXException
```

Notice that nowhere in this method is there any information as to which element these characters are part of. If you need this information, you're going to have to store it. This example adds variables to store the current element and question information. (It also removes a lot of extraneous information that was displayed.)

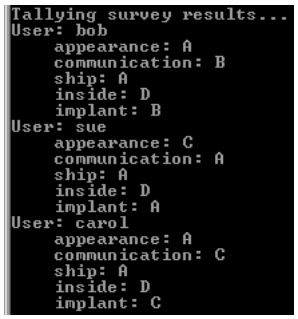
There are two important things to note here:

Range: That the `characters()` event includes more than just a string of characters. It also includes start and length information. In actuality, the `ch` character array includes the entire

document. The application must not attempt to read characters outside the range the event feeds to the `characters()` event.

Frequency: Nothing in the SAX specification requires a processor to return characters in any particular way, so its possible for a single chunk of text to be returned in several pieces. Always make sure that the endElement() on page 15 event has occurred before assuming you have all the content of an element. Also, processors may use ignorableWhitepsace() on page 18to return whitespace within an element. This is always the case for a validating parser.

```
...
public void printIndent(int indentSize) {
   for (int s = 0; s < indentSize; s++) {
      System.out.print(" ");
   }
}

String thisQuestion = "";
String thisElement = "";
public void startElement(
                String namespaceURI,
                String localName,
                String qName,
                Attributes atts)
                   throws SAXException {

   if (localName == "response") {
      System.out.println("User: "
           + atts.getValue("username"));
   } else if (localName == "question") {
      thisQuestion = atts.getValue("subject");
   }

   thisElement = localName;
}

public void endElement(
                String namespaceURI,
                String localName,
                String qName)
                    throws SAXException {
   thisQuestion = "";
   thisElement = "";
}

public void characters(char[] ch,
                       int start,
                       int length)
                  throws SAXException  {

   if (thisElement == "question") {
      printIndent(4);
      System.out.print(thisQuestion + ": ");
      System.out.println(new String(ch,
                                    start,
                                    length));
   }
}
...
```



Record the answers

Now that you've got the data, go ahead and add the actual tally.

This is as simple as building strings for analysis when the survey is complete.

```
...
 public void characters(char[] ch,
                        int start,
                        int length)
                        throws SAXException  {

    if (thisElement == "question") {

       if (thisQuestion.equals("appearance")) {
          appearance = appearance + new String(ch, start, length);
       }
       if (thisQuestion.equals("communication")) {
          communication = communication + new String(ch, start, length);
       }
       if (thisQuestion.equals("ship")) {
          ship = ship + new String(ch, start, length);
       }
       if (thisQuestion.equals("inside")) {
          inside = inside + new String(ch, start, length);
       }
       if (thisQuestion.equals("implant")) {
          implant = implant  + new String(ch, start, length);
       }

    }
 }
 ...
```

ignorableWhitepsace()

XML documents produced by humans (as opposed to programs) often include whitespace added to make the document easier to read. Whitespace includes line breaks, tabs, and spaces. In most cases, this whitespace is extraneous and should be ignored when the data is processed.

All validating parsers, and some non-validating parsers, pass these whitespace characters to the content handler not in `characters()`, but in the `ignorableWhitespace()` event. This is convenient, because you can then concentrate only on the actual data.

But what if you really *want* the whitespace? In such a scenario, you set an attribute on the element that signals the processor not to ignore whitespace characters. This attribute is

`xml:space`, and it is usually assumed to be `default`. (This means that unless the default behavior of the processor is to preserve whitespace, it will be ignored.)

To tell the processor *not* to ignore whitespace, set this value to `preserve`, as in:

```
<code-snippet xml:space="preserve">
    public void endElement(
                String namespaceURI,
                String localName,
                String qName)
                    throws SAXException
 </code-snippet>
```

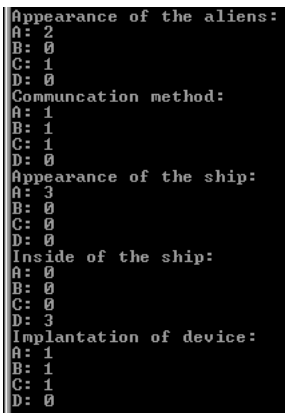
---

endDocument()

And of course, once the document is completely parsed, you'll want to print out the final tally as shown below. (The actual tally method, `getInstances()`, is irrelevant.)

This is also a good place to tie up any loose ends that may have come up during processing.

```
...
       if (thisQuestion.equals("implant")) {
          implant = implant
               + new String(ch, start, length);
       }
    }
 }

 public int getInstances (String all,
                          String choice) {
    ...
    return total;
 }

 public void endDocument() {

    System.out.println("Appearance of the aliens:");
    System.out.println("A: " +
                       getInstances(appearance, "A"));
    System.out.println("B: " +
                       getInstances(appearance, "B"));
    ...
 }

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



---

processingInstruction()

All of this is well and good, but sometimes we want to include information directly for the application that's processing the data. One good example of this is when we want to process a document with a style sheet, as in:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="survey.xsl" version="1.0" type="text/xsl" ?>
<surveys>
...
```

Other applications can get information in similar ways. For instance, we certainly don't have a large enough statistical sample to be taken seriously. We could add a processing instruction just for our application that specifies a factor to multiply the responses by:

```
<?xml version="1.0" encoding="UTF-8"?>
<?SurveyReader factor="2" ?>
<surveys>
...
```

This would be picked up by the `processingInstruction()` event, which separates it into the `target` and the `data`.

For simplicity, we'll just echo back the information for now:

```
...
public void processingInstruction(String target, String data)
      throws SAXException {

   System.out.println("Target = ("+target+")");
   System.out.println("Data = ("+data+")");
}
...
```

```
Tallying survey results...
Target = (SurveyReader)
Data = (factor="2")
Appearance of the aliens:
```

Section 5. SAX and Namespaces

Namespaces

One of the main enhancements that was added to SAX version 2.0 is the addition of support for *Namespaces*. Namespace support allows developers to use information from different sources or with different purposes without conflicts. This often arises in a production environment, where data in the SAX stream can come from many different sources.

Namespaces are conceptual zones in which all names need to be unique.

For example, I used to work in an office where I had the same first name as a client. If I were in the office and the receptionist announced "Nick, pick up line 1", everyone knew she meant me, because I was in the "office namespace". Similarly, if she announced "Nick is on line 1", everyone knew that it was the client, because whomever was calling was outside the office namespace.

On the other hand, if I were out of the office and she made the same announcement, there would be confusion, because two possibilities would exist.

The same issues arise when XML data is combined from different sources (such as the revised survey information in the sample file detailed later in this tutorial).

Creating a namespace

Because identifiers for namespaces must be unique, they are designated with Uniform Resource Identifiers, or URIs. For example, a default namespace for the sample data would be designated using the `xmlns` attribute:

```
<?xml version="1.0"?>
<surveys xmlns="http://www.nicholaschase.com/surveys/">
<response username="bob">
   <question subject="appearance">A</question>
...
```

(The `...` indicates sections that aren't relevant.)

Any nodes that don't have a namespace specified are in the default namespace, http://www.nicholaschase.com/surveys/. The actual URI itself doesn't mean anything. There may or may not be information at that address, but what is important is that it is unique.

Secondary namespaces can also be created, and elements or attributes added to them.

Designating namespaces

Other namespaces can also be designated for data. For example, add a second set of data -- say, post-hypnosis -- can be added without disturbing the original responses by creating a `revised` namespace.

The namespace, along with an alias, is created, usually (but not necessarily) on the document's root element. This alias is used as a prefix for elements and attributes -- as necessary, when more than one namespace is in use -- to specify the correct namespace. Consider the code below.

```
<?xml version="1.0"?>
<surveys xmlns="http://www.nicholaschase.com/surveys/"
         xmlns:rating="http://www.nicholaschase.com/surveys/revised/">
<response username="bob">
   <question subject="appearance">A</question>
   <question subject="communication">B</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
   <question subject="implant">B</question>
   <revised:question subject="appearance">D</revised:question>
   <revised:question subject="communication">A</revised:question>
   <revised:question subject="ship">A</revised:question>
   <revised:question subject="inside">D</revised:question>
   <revised:question subject="implant">A</revised:question>
</response>
<response username="sue">
...
```

The namespace and the alias, `revised`, have been used to create an additional `question` element.

Checking for namespaces

Version 2.0 of SAX adds functionality for recognizing different namespaces, as mentioned briefly when discussing startElement() on page 13 .

There are several ways to use these new abilities, but start by making sure that only the original answers come up in the results. Otherwise, Bob's answers are going to count twice.

Because the answer won't be recorded unless `thisElement` is `"question"`, simply do the check before setting that variable.

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

   if (namespaceURI ==
     "http://www.nicholaschase.com/surveys/") {

      if (localName == "question") {
         thisQuestion = atts.getValue("subject");
      }

   }

   thisElement = localName;
}
...
```

Note that the application is actually looking at the namespace URI (or URL, in this case), as opposed to the alias.

Namespaces on attributes

Attributes can also belong to a particular namespace. For instance, if the name of the questions changed the second time around, you could add a second related attribute, `revised:subject`, like so:

```
<?xml version="1.0"?>
 <surveys xmlns="http://www.nicholaschase.com/surveys/"
          xmlns:revised="http://www.nicholaschase.com/surveys/revised/">
 <response username="bob">
    <question subject="appearance">A</question>
    <question subject="communication">B</question>
    <question subject="ship">A</question>
    <question subject="inside">D</question>
    <question subject="implant">B</question>
    <revised:question subject="appearance" revised:subject="looks">D</revised:question>
    <revised:question subject="communication">A</revised:question>
    <revised:question subject="ship">A</revised:question>
    <revised:question subject="inside">D</revised:question>
    <revised:question subject="implant">A</revised:question>
 </response>
 <response username="sue">
 ...
```

The `Attributes` list has methods that allow you to determine the namespace of an attribute. These methods, `getURI()` and `getQName`, are used much like the `qname` and `localName` for the element itself.

Section 6. A SAX stream as output

Serializing a SAX stream

Examples so far have looked at the basics of working with data, but this is just a simple look at what SAX can do. Data can be directed to another SAX process, a transformation, or of course, to a file.

Outputting data to a file is known as *serializing* the data.

You could construct a data file manually, but that's a lot of unnecessary work. Instead, pass the data to a `Serializer` as the content handler. `Serializer` is part of Xalan, which is normally used to do XSL transformations.

As the events pass to the `serializer`, which has been designated as the content handler, it outputs the data to its `OutputStream`.

```
                    import org.apache.xalan.serialize.Serializer;
import org.apache.xalan.serialize.SerializerFactory;
import org.apache.xalan.templates.OutputProperties;
...
public static void main (String args[]) {

   XMLReader xmlReader = null;

   try {

      SAXParserFactory spfactory =
             SAXParserFactory.newInstance();
      spfactory.setValidating(false);
      SAXParser saxParser =
                     spfactory.newSAXParser();

      xmlReader = saxParser.getXMLReader();
      Serializer serializer =
           SerializerFactory.getSerializer(
               OutputProperties
                .getDefaultMethodProperties("xml"));
      serializer.setOutputStream(System.out);
      xmlReader.setContentHandler(
                    serializer.asContentHandler());

      InputSource source =
                    new InputSource("surveys.xml");
      xmlReader.parse(source);

   } catch (Exception e) {
         System.err.println(e);
         System.exit(1);
   }
}
...
```

```
<?xml version="1.0" encoding="UTF-8" standalone="n
<surveys>
<response username="bob">
   <question subject="appearance">A</question>
   <question subject="communication">B</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
   <question subject="implant">B</question>
   <revised:question subject="appearance" revised:
stion>
   <revised:question subject="communication">A</re
   <revised:question subject="ship">A</revised:ques
   <revised:question subject="inside">D</revised:qu
   <revised:question subject="implant">A</revised:
</response>
<response username="sue">
   <question subject="appearance">C</question>
   <question subject="communication">A</question>
   <question subject="ship">A</question>
   <question subject="inside">D</question>
```

XMLFilters

Because SAX involves analyzing data (as opposed to storing it) as it passes by, you may think that there's no way to alter the data before it's analyzed.

This is the problem that `XMLFilters` solve. Although they're new to version 2.0 of SAX, they were actually in use in version 1.0 by clever programmers who realized that they could "chain" SAX streams together, effectively manipulating them before they reached their final destination.

Basically, it works like this:

1. Create the `XMLFilter`. This is typically a separate class.
2. Create an instance of the `XMLFilter` and set its parent to be the `XMLReader` that would normally parse the file.
3. Set the content handler of the filter to be the normal content handler.
4. Use the filter to parse the file. The filter acts on the events first, then passes them on to the `XMLReader`, and then it acts on the events using the normal content handler.

Creating a filter

What we want to do now is create a filter that allows us to discard the user's original answers and instead use the revised answers. To do this, we need to "erase" the originals, then change the namespace on the revised answers so `SurveyReader` can pick them up.

This is implemented by creating a new class that extends `XMLFilterImpl`.

Let's look at what's happening here. When the `startElement()` event fires, it checks the original namespaceURI. If this is a revised element, the namespace is changed to the default namespace. If it's not, the element name is actually changed so that the original tally routine (in `SurveyReader`) won't recognize it as a question, and consequently, does not count the answer.

This altered data is passed on to the `startElement()` for our parent, the original `XMLReader`, so it's taken care of by the content handler.

```
import org.xml.sax.helpers.XMLFilterImpl;
 import org.xml.sax.XMLReader;
 import org.xml.sax.SAXException;
 import org.xml.sax.Attributes;

 public class SurveyFilter extends XMLFilterImpl
 {
   public SurveyFilter ()
   {
   }

   public SurveyFilter (XMLReader parent)
   {
     super(parent);
   }

   public void startElement (String uri,
                             String localName,
```

```
                            String qName,
                            Attributes atts)
     throws SAXException
   {
       if (uri == "http://www.nicholaschase.com/surveys/revised/") {
           uri = "http://www.nicholaschase.com/surveys/";
           qName = localName;
       } else {
           localName = "REJECT";
       }

       super.startElement(uri, localName, qName, atts);
   }

 }
```

Invoking the filter

Now it's time to use the filter. The first thing to do is create a new instance, then assign the original `XMLReader` as its parent.

Next, set the content and error handlers on the filter, as opposed to the reader. Finally, use the filter to parse the file, instead of the `XMLReader`.

Because the `XMLReader` is designated as the filter's parent, it still processes the information.

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

    XMLReader xmlReader = null;

    try {

       SAXParserFactory spfactory =
              SAXParserFactory.newInstance();
       spfactory.setValidating(false);
       SAXParser saxParser =
                    spfactory.newSAXParser();

       xmlReader = saxParser.getXMLReader();
       SurveyFilter xmlFilter = new SurveyFilter();
       xmlFilter.setParent(xmlReader);

       xmlFilter.setContentHandler(new SurveyReader());
       xmlFilter.setErrorHandler(new SurveyReader());

       InputSource source = new InputSource("surveys.xml");
       xmlFilter.parse(source);

       } catch (Exception e) {
             System.err.println(e);
             System.exit(1);
       }
    }
 ...
```

There is no limit to how deeply these features can be nested. Theoretically, you could create a long chain of filters, each one calling the next.

Using an XMLFilter to transform data

`XMLFilters` can also be used to quickly and easily transform data using XSLT.

The transformation itself is beyond the scope of this tutorial, but take a look at how you would apply it.

The example combines two different techniques. First, create the filter. But instead of creating it from scratch, create a filter that is specifically designed to execute a transformation based on a style sheet.

Then, just as you did when you were outputting the file directly, create a `Serializer` to output the result of the transformation.

Basically, the filter performs the transformation, then hands the events on to the `XMLReader`. Ultimately, however, the destination is the serializer.

```
                    import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.sax.SAXTransformerFactory;
import org.xml.sax.XMLFilter;
...
public static void main (String args[]) {

   XMLReader xmlReader = null;

   try {

      SAXParserFactory spfactory =
            SAXParserFactory.newInstance();
      spfactory.setValidating(false);
      SAXParser saxParser =
                     spfactory.newSAXParser();

      xmlReader = saxParser.getXMLReader();

      TransformerFactory tFactory =
                    TransformerFactory.newInstance();
      SAXTransformerFactory saxTFactory =
                  ((SAXTransformerFactory) tFactory);

      XMLFilter xmlFilter =
               saxTFactory.newXMLFilter(
                     new StreamSource("surveys.xsl"));
      xmlFilter.setParent(xmlReader);

      Serializer serializer =
             SerializerFactory.getSerializer
                (OutputProperties
                   .getDefaultMethodProperties("xml"));
      serializer.setOutputStream(System.out);
      xmlFilter.setContentHandler(
               serializer.asContentHandler());

      InputSource source =
               new InputSource("surveys.xml");
      xmlFilter.parse(source);

   } catch (Exception e) {
         System.err.println(e);
         System.exit(1);
   }
}
...
```

Section 7. SAX summary

Summary

SAX is a faster, more lightweight way to read and manipulate XML data than the Document Object Model (DOM). SAX is an event-based processor that allows you to deal with elements, attributes, and other data as it shows up in the original document. Because of this architecture, SAX is a read-only system, but that doesn't prevent you from using the data. We also looked at ways to use SAX to determine Namespaces, and to output and transform data using XSL. And finally, we looked at Filters, which allow you to chain operations.

Resources

Information on XML in general and SAX in particular is exploding on the Web. Here are some good places to start:

For a basic grounding in XML read through the *Introduction to XML* tutorial. See this page for information covering the *SAX API* , especially SAX2. *The Collected Works of SAX* by Leigh Dodds provides interesting coverage of SAX and its developments. Read details about the results of JavaTM *parsers* tested using the SAX API. Read *Benoit Marchal's SAX preview* from the second edition of *XML by Example*. Read Brett McLaughlin's tip on *how to move from SAX into DOM and JDOM* . Order *XML and Java from Scratch* , by Nicholas Chase, the author of this tutorial. Download *the Java 2 SDK* , Standard Edition version 1.3.1. Download *JAXP 1.1* , the JavaTM APIs for XML Processing.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.