

Understanding SAX

Skill Level: Introductory

[Nicholas Chase \(nicholas@nicholaschase.com\)](mailto:nicholas@nicholaschase.com)

Author, Web site developer

29 Jul 2003

This tutorial examines the use of the Simple API for XML version 2.0.x, or SAX 2.0.x. 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. In this tutorial, you will learn how to use SAX to retrieve, manipulate, and output XML data.

Section 1. Tutorial introduction

Should I take this tutorial?

This tutorial examines the use of the Simple API for XML version 2.0.x, or SAX 2.0.x. 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 the Java language in its demonstrations, but the concepts are substantially similar in all languages, and you can gain a thorough understanding of SAX without actually working through the examples.

What is SAX?

The standard means for reading and manipulating XML files is the Document Object Model (DOM). Unfortunately this method, which involves reading the entire file and storing it in a tree structure, can be inefficient, slow, and 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 now a SourceForge project (see [Resources](#)). The purpose of the project was to provide a more natural means for working with XML -- in other words, one that did not involve the overhead and conceptual leap required for 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.

SAX has no official standards body; 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.
- **Java™ 2 SDK, Standard Edition version 1.4.x:** SAX support has been built into the latest version of Java (available at <http://java.sun.com/j2se/1.4.2/download.html>), so you won't need to install any separate classes. If you're using an earlier version of Java, such as Java 1.3.x, you'll also need an XML parser such as the Apache project's Xerces-Java (available at <http://xml.apache.org/xerces2-j/index.html>), or Sun's Java API for XML Parsing (JAXP), part of the Java Web Services Developer Pack (available at <http://java.sun.com/webservices/downloads/webservicespack.html>). You can also download the official version from SourceForge (available at

http://sourceforge.net/project/showfiles.php?group_id=29449).

- **Other Languages:** Should you wish to adapt the examples, SAX implementations are also available in other programming languages. You can find information on C, C++, Visual Basic, Perl, and Python implementations of a SAX parser at <http://www.saxproject.org/?selected=langs>.

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 ticker tape. Consider the following XML code snippet:

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

A SAX processor analyzing this code snippet 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:

1. Create an event handler.
2. Create the SAX parser.

3. Assign the event handler to the parser.
4. 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 waiting 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 fact, an application doesn't even have to parse the entire document; it can stop when certain criteria have been satisfied. In general, SAX is also much faster than the alternative, the DOM.

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 backwards in the data stream.

DOM and tree-based processing

The 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 [How SAX processing works](#) would be represented as nodes, shown here:

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

DOM uses parent-child relationships. For instance, in this case `samples` is the root element with five children: three text nodes (the whitespace), and the two element nodes, `server` and `monitor`.

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

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, a lot of overhead is 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. It's not that you can't make changes using SAX, but the process is much more complex, as you'd have to make changes to a copy of the data rather than to the data itself.
- **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 large amounts of 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 stream of SAX 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.

Here is the survey form:

The responses are stored in an XML file:

```
<?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.

Proper parsing using SAX requires specific calls to the handler's methods, and these calls *can't be static*. This means you need to specifically instantiate the handler object, so I'll give a quick overview of that, in case you're not used to working with objects.

When a new object is created, it looks for any class constructors to execute. For example:

```
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");
    }
}
```

When the `main` method executes, it creates a new instance of the `SurveyReader` class. This causes the constructor to execute, outputting `Object Created` (along with the greeting below). You can then use that object to execute the `showEvent()` method.

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. You can create the parser 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 were (the nonexistent) `com.nc.xml.SAXDriver`, you could use the code:

```
try {
    XMLReader xmlReader = new com.nc.xml.SAXDriver();
} catch (Exception e) {
    System.out.println("Can't create the parser: " + e.getMessage());
}
```

to directly create the `XMLReader`.

You can also use system properties to make your application more flexible. For example, you can specify the class name as the value for the `org.xml.sax.driver` property from the command line when you run the application:

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

(No, `-D` is *not* supposed to have a space after it.)

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

```
try {
    XMLReader xmlReader = XMLReaderFactory.createXMLReader();
} catch (Exception e) {
    System.out.println("Can't create the parser: " + e.getMessage());
}
```

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

Create the parser

This example uses a pair of classes, `SAXParserFactory` and `SAXParser`, to create the parser, so you don't have to know the name of the driver itself.

First declare the `XMLReader`, `xmlReader`, and then 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 versus non-validating parsers

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 two kinds of parsers are: *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 -- or in the case of a SAX application, processes the event -- without regard for the actual structure or content.
- A **validating parser**, on the other hand, checks the content and structure

of an XML document against a defined grammar. Sometimes this grammar is in the form of a Document Type Definition (DTD), but these days it's more likely to be defined in an XML Schema document. 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, you might specify that every `order` must have a `status`. If you 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

For this tutorial, you're not validating the survey results, so turn off validation for any parser the `SAXParserFactory` creates by setting the `validating` property:

```
...
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

The parser has to send its events to a `ContentHandler`. To keep things simple, `SurveyReader` is both the main application and the content handler, so create a new instance of it and set it as the `ContentHandler` using the `XMLReader`'s `setContentHandler()` method:

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

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

This is not, of course, the only option for a content handler. Later in this tutorial, you'll see other options when this tutorial examines [Serializing a SAX stream](#).

Parse the InputSource

To actually parse a file (or anything else, for that matter!) you need an `InputSource`. 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 `parse()` method takes the file, wrapped in the `InputSource`, and processes it, sending each event to the `ContentHandler`.

```
...  
  
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.

Set an ErrorHandler

At this point nothing *should* happen, but there is, of course, always the chance of 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, you'll include error handling right in `SurveyResults`. This dual-usage is possible because the class extends `DefaultHandler`, which includes implementations of both the `ContentHandler` methods and the `ErrorHandler` methods.

Set a new `ErrorHandler` just as you set the `ContentHandler`:

```
...
xmlReader.setContentHandler(new SurveyReader());
xmlReader.setErrorHandler(new SurveyReader());
InputSource source = new InputSource("surveys.xml");
...
```

Still, nothing should happen if you run the application because the default implementations for each event don't do anything. In the next section, I'll look at adding new implementations to handle the events that occur during parsing.

Section 4. Event handlers and the SAX events

startDocument()

Now that you're set up to parse the document, it's time to start replacing the default implementations that are part of the `DefaultHandler` class with methods that actually do something when the handler receives the appropriate event.

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 SurveyReader() {
    }

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

    public static void main (String args[]) {
        XMLReader xmlReader = null;
    }
}
```

startElement()

Now, begin looking at the actual data. For each element, the example echoes back

the name that is passed to the `startElement()` event (see the Element Listing screenshot below).

The parser actually passes several pieces of information for each element:

- **The qualified name, or `qName`.** This is actually a combination of namespace information, if any, and the actual name of the element. The `qName` also includes the colon (:) if there is one -- for example, `revised:response`.
- **The namespace URI.** As discussed in [Namespaces](#), an actual namespace is a URI of some sort and not the alias that gets added to an element or attribute name. For example, `http://www.nicholaschase.com/surveys/revised/` as opposed to simply `revised:`.
- **The local name.** This is the actual name of the element, such as `question`. If the document doesn't provide namespace information, the parser may not be able to determine which part of the `qName` is the `localName`.
- **Any attributes.** The attributes for an element are actually passed as a collection of objects, as seen in the next panel.

Start by listing the name of each 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(qName);  
  
    }  
  
    public static void main (String args[]) {  
        ...  
    }  
}
```

`startElement()`: retrieve attributes

The `startElement()` event also provides access to the attributes for an element. They are passed in within an `Attributes` data structure.

You can retrieve an attribute value based on its position in the array, or based on the name of the attribute:

```
...  
  
public void startElement(String namespaceURI, String localName,  
                        String qName, Attributes atts) throws SAXException {  
  
    System.out.print("Start element: ");  
    System.out.println(qName);  
  
    for (int att = 0; att < atts.getLength(); att++) {  
        String attName = atts.getQName(att);  
        System.out.println("    " + attName + ": " + atts.getValue(attName));  
    }  
  
}  
...  
}
```

endElement()

You'll find plenty of good reasons to note the end of an element. For example, it might be a signal to process the contents of an element. Here you'll use it to **pretty print** the document to some extent, indenting each level of elements. The idea is to 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(qName);  
  
    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;
}

...

```

characters()

Now that you've got the elements, 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 whatsoever as to what 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.)

Note two important things here:

- **Range:** 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 it's possible for a single chunk of text to be returned in several pieces. Always make sure that the `endElement()` event has occurred before assuming you have all the content of an element. Also, processors may use `ignorableWhitespace()` to 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 (qName == "response") {
        System.out.println("User: " + atts.getValue("username"));
    } else if (qName == "question") {
        thisQuestion = atts.getValue("subject");
    }

    thisElement = qName;
}

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.

```

...

String appearance = null;
String communication = null;
String ship = null;
String inside = null;
String implant = null;

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);
    }
}
}
...

```

It's important to remember that the only reason you can perform this task in the `characters()` method is because the answers you're looking for are only one character long. If the content you were looking for were any longer, you would have to gather the data from each call together and analyze it at the end of the element, in the `endElement()` method.

ignorableWhitespace()

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 do *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">
  <line> public void endElement(</line>
    <line>     String namespaceURI,</line>
    <line>     String localName,</line>
    <line>     String qName)</line>
    <line>     throws SAXException</line>
</code-snippet>

```

endDocument()

And of course, once the document is completely parsed, you'll want to print out the final tally as shown below.

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 you may want to include information directly for the application that's processing the data. One good example of this is when you want to process a document with a stylesheet, 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, a survey has a statistical sample that certainly isn't large enough to be taken seriously. You could add a processing instruction just for your 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`:

```
...
public void processingInstruction(String target, String data)
    throws SAXException {
    System.out.println("Target = (" +target+")");
    System.out.println("Data = (" +data+")");
}
...
```

ErrorHandler events

Just as the `ContentHandler` has predefined events for handling content, the `ErrorHandler` has predefined events for handling errors. Because you specified `SurveyReader` as the error handler as well as the content handler, you need to override the default implementations of those methods.

You need to be concerned with three events: `warning`, `error`, and `fatalError`:

```
...

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);
    }
}

...
```

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**, which allow 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 the caller was outside the office namespace.

On the other hand, if I were out of the office and she made the same announcement, confusion would be likely because two possibilities 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 (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>
  ...
```

Any nodes without a namespace specified are in the default namespace, `http://www.nicholaschase.com/surveys/`. The actual URI itself doesn't mean anything. Information may or may not be at that address, but what is important is that it is unique.

It's important to note that a huge distinction exists between the default namespace and no namespace at all. In this case, elements without a namespace prefix are in the default namespace. Previously, when no default namespace existed, those elements were in no namespace. This distinction becomes important when you deal with [Namespaces on attributes](#).

You can also create secondary namespaces, and add elements or attributes to them.

Designating namespaces

Other namespaces can also be designated for data. For example, 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 the 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 following code:

```
<?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">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.

Remember that `revised:` is not the namespace, but an alias! The actual namespace is `http://www.nicholaschase.com/surveys/revised/`.

Checking for namespaces

Version 2.0 of SAX adds functionality for recognizing different namespaces, as

mentioned briefly in [startElement\(\)](#).

You can use these new abilities in several ways, 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", you should 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">
...

```

One slightly odd thing about attributes is that they are *never* in the default namespace. Even if a default namespace has been declared, an attribute without a prefix is considered to have no namespace at all. This means that `response` is in the `http://www.nicholaschase.com/surveys/` namespace, but its attribute `username` is not. This is just an oddity defined in the XML Recommendation itself.

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.

Namespace oddities

The manner in which a SAX parser handles local names and QNames can be a little odd. For example, the default Java parser won't report local name values unless the namespace process is specifically turned on:

```
...
try {
    SAXParserFactory spfactory = SAXParserFactory.newInstance();
    spfactory.setValidating(false);
    spfactory.setFeature("http://xml.org/sax/features/namespace-prefixes",
        true);
    spfactory.setFeature("http://xml.org/sax/features/namespaces",
        true);

    SAXParser saxParser = spfactory.newSAXParser();
...

```

If you have difficulty getting to information, try setting these features for the parser.

Section 6. Working with a SAX stream

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. In this section, I'll show you some of these

options.

Outputting the stream of SAX events to a file is called **serializing** it. You can write the file yourself, but it's so much easier to simply use the `Serializer` object:

```
import org.apache.xalan.serialize.Serializer;
import org.apache.xalan.serialize.SerializerFactory;
import org.apache.xalan.templates.OutputProperties;
import java.io.FileOutputStream;

...
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(new FileOutputStream("output.xml"));

        xmlReader.setContentHandler(
serializer.asContentHandler()
    );

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

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

Create the `Serializer` object -- acceptable values for the `OutputProperties` are `xml`, `text`, and `html` -- and set its `OutputStream`. This stream can be virtually any stream-type object, such as a file (as it is here) or `System.out`.

You can then set the `Serializer` as the content handler for the parser, so when the parser parses the file, it's the `Serializer` that receives the events.

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 `XMLFilter`s 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. Parse the file. The filter sits between the `XMLReader` and the content handler.

Creating a filter

Now you want to create a filter that allows you to discard the user's original answers and instead use the revised answers. To do this, you need to erase the originals, and 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`.

Take a look at what's happening here. When the `startElement()` event fires, it checks the original namespace URI. 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 the 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 = "question";
        } 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);
    }
}

```

```
    }  
    ...
```

The depth of nesting for these features is unlimited. 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 here's a quick look at how you would apply it:

```
        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.xml"));  
  
        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);  
    }  
}
```

First, you need to create the filter -- but instead of creating it from scratch, create a filter that is specifically designed to execute a transformation based on a stylesheet.

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.

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. This tutorial also looked at ways to use SAX to determine namespaces, and to output and transform data using XSL. And finally, you looked at filters, which allow you to chain operations.

Resources

Learn

- For a basic grounding in XML read through the "[Introduction to XML](#)" tutorial (*developerWorks*, August 2002).
- See the official [SAX 2.0](http://www.saxproject.org) page (<http://www.saxproject.org>).
- Read about using [SAX filters for flexible processing](#) (*developerWorks*, March 2003).
- Read about using [SAX filters for flexible processing](#) (*developerWorks*, March 2003).
- Find out how to build [SAX-like apps in PHP](#) (*developerWorks*, March 2003).
- Learn how to [set up a SAX parser](#) (*developerWorks*, July 2003).
- Learn more about [validation and the SAX ErrorHandler interface](#) (*developerWorks*, June 2001).
- Understand how to [stop a SAX parser when you have enough data](#) (*developerWorks*, June 2002).
- Explore [XSL transformations to and from a SAX stream](#) (*developerWorks*, July 2002).
- Find out how to turn a SAX stream into a DOM or JDOM object with "[Converting from SAX](#)" (*developerWorks*, April 2001).
- Order [XML Primer Plus](#), by Nicholas Chase, the author of this tutorial (<http://www.amazon.com/exec/obidos/ASIN/0672324229/ref=nosim/thevanguardsc-20>).
- SAX was developed by the members of the XML-DEV mailing list. Try the Java version, now a [SourceForge project](#) (http://sourceforge.net/project/showfiles.php?group_id=29449).
- Try SAX implementations that are also available in [other programming languages](#) (<http://www.saxproject.org/?selected=langs>).
- Find out how you can become an [IBM Certified Developer in XML and related technologies](#) (<http://www-1.ibm.com/certify/certs/adcdxmlrt.shtml>).

Get products and technologies

- Download [the Java 2 SDK](#), Standard Edition version 1.4.2 (<http://java.sun.com/j2se/1.4.2/download.html>).
- If you're using an older version of Java, download the Apache project's [Xerces-Java](#) (<http://xml.apache.org/xerces2-j/index.html>), or Sun's Java API for XML Parsing (JAXP), part of the [Java Web Services Developer Pack](#)

(<http://java.sun.com/webservices/downloads/webservicespack.html>).

- Get IBM's [DB2](http://www.ibm.com/software/data/db2/) database (<http://www.ibm.com/software/data/db2/>) which provides not only relational database storage, but also XML-related tools such as the [DB2 XML Extender](http://www.ibm.com/software/data/db2/extenders/xmlxt/) (<http://www.ibm.com/software/data/db2/extenders/xmlxt/>) which provides a bridge between XML and relational systems. Visit the [DB2 content area](#) to learn more about DB2 (<http://www.ibm.com/developerworks/db2/>).

About the author

Nicholas Chase

Nicholas Chase, a [Studio B](#) author, has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, 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, USA, and is the author of four books on Web development, including *XML Primer Plus* (Sams). He loves to hear from readers and can be reached at nicholas@nicholaschase.com.