XML programming in Java technology, Part 1

Skill Level: Introductory

Doug Tidwell (dtidwell@us.ibm.com)
XML Evangelist

13 Jan 2004

This updated tutorial covers the basics of manipulating XML documents using Java technology. Doug Tidwell looks at the common APIs for XML and discusses how to parse, create, manipulate, and transform XML documents.

Section 1. Introduction

About this tutorial

Over the last few years, XML has become a universal data format. In this updated tutorial, I'll show you the most common programming interfaces for working with XML documents in the Java language.
The most common XML processing task is **parsing** an XML document. Parsing involves reading an XML document to determine its structure and contents. One of the pleasures of XML programming is the availability of open-source, no-cost XML parsers that read XML documents for you. This tutorial focuses on creating parser objects, asking those parsers to process XML files, and handling the results. As you might expect, you can do these common tasks in several different ways; I'll examine the standards involved as well as when you should use one approach or another.

Programming interfaces

A number of programming interfaces have been created to simplify writing Java programs that process XML. These interfaces have been defined by companies, by standards bodies, and by user groups to meet the needs of XML programmers. In this tutorial, I'll cover the following interfaces:

- The Document Object Model (DOM), Level 2
- The Simple API for XML (SAX), Version 2.0

- JDOM, a simple Java API created by Jason Hunter and Brett McLaughlin
- The Java API for XML Processing (JAXP)

The first three of these four interfaces (DOM, SAX, and JDOM) define how the contents of an XML document are accessed and represented. JAXP contains classes for creating parser objects. To create DOM or SAX parsers, you'll use JAXP. When you use JDOM, the JDOM library uses JAXP under the covers to create a parser. To sum it all up:

- You use DOM, SAX, or JDOM to work with the contents of an XML document.
- If you use DOM or SAX, you use JAXP to create a parser.
- If you use JDOM, the JDOM library creates a parser for you.

I'll explore the design goals, strengths, and weaknesses of each of these APIs, along with a bit of their histories and the standards bodies that created them.

About the examples

Throughout this tutorial, I'll show you a number of sample programs that use the DOM, SAX, and JDOM APIs. All of them work with an XML-tagged Shakespearean sonnet. The structure of the sonnet is:

```
 <sonnet>
  <author>
    <lastName>
    <firstName>
    <nationality>
    <yearOfBirth>
    <yearOfDeath>
  </author>
  <lines>
    [14 <line> elements]
  </lines>
</sonnet>
```

For the complete example, see sonnet.xml and sonnet.dtd (download to view in a text editor).

Setting up your machine

You'll need to set up a few things on your machine before you can run the examples. (I'm assuming that you know how to compile and run a Java program, and that you know how to set your `CLASSPATH` variable.)

1. First, visit the home page of the Xerces XML parser at the Apache XML Project (http://xml.apache.org/xerces2-j/). You can also go directly to the download page (http://xml.apache.org/xerces2-j/download.cgi).
2. Unzip the file that you downloaded from Apache. This creates a directory

named `xerces-2_5_0` or something similar, depending on the release level of the parser. The JAR files you need (`xercesImpl.jar` and `xml-apis.jar`) should be in the Xerces root directory.

3. Visit the JDOM project's Web site and download the latest version of JDOM (http://jdom.org/).

4. Unzip the file you unloaded from JDOM. This creates a directory named `jdom-b9` or something similar. The JAR file you need (`jdom.jar`) should be in the `build` directory.

5. Finally, download the zip file of examples for this tutorial, xmlprogj.zip, and unzip the file.

6. Add the current directory (`.`), `xercesImpl.jar`, `xml-apis.jar`, and `jdom.jar` to your `CLASSPATH`.

Section 2. Parser basics

The basics

An XML parser is a piece of code that reads an XML document and analyzes its structure. In this section, you'll see how XML parsers work. I'll show you the different kinds of XML parsers and when you might want to use them.
Later sections of this tutorial will discuss how to create parsers and how to process the results they give you.

How to use a parser

I'll talk about this in much more detail in future sections, but in general, here's how you use a parser:

1. Create a parser object

2. Point the parser object at your XML document

3. Process the results

Obviously the third step is the complicated one. Once you know the contents of the XML document, you might want to, for example, generate a Web page, create a purchase order, or build a pie chart. Considering the infinite range of data that could be contained in an XML document, the task of writing an application that correctly processes any potential input is intimidating. Fortunately, the common XML parsing tools discussed here can make the task much, much simpler.

Kinds of parsers

Parsers are categorized in several different ways:

- Validating versus non-validating parsers
- Parsers that support one or more XML Schema languages
- Parsers that support the Document Object Model (DOM)
- Parsers that support the Simple API for XML (SAX)

Validating versus non-validating parsers

For those of you who are new to XML, there are three different kinds of XML documents:

- **Well-formed documents:** These follow the basic rules of XML (attributes must be quoted, tags must be nested correctly, and so forth).
- **Valid documents:** These are well-formed documents that also conform to a set of rules specified in a Document Type Definition (DTD) or an XML Schema.
- **Invalid documents:** Everything else.

For example, if you have an XML document that follows all of the basic rules of XML, it's a *well-formed* document. If that document also follows all of the rules you've defined for expense account documents in your company, it's *valid* as well.

If an XML parser finds that an XML document is not well-formed, the XML Specification requires the parser to report a fatal error. Validation, however, is a different issue. **Validating parsers** validate XML documents as they parse them, while **non-validating parsers** don't. In other words, if an XML document is well-formed, a non-validating parser doesn't care if that document follows the rules defined in a DTD or schema, or even if there are any rules for that document at all. (Most validating parsers have validation turned off by default.)

Why use a non-validating parser?

So why use a non-validating parser? Two good reasons: Speed and efficiency. It takes a significant amount of effort for an XML parser to read a DTD or schema, then set up a rules engine that makes sure every element and attribute in an XML document follows the rules. If you're sure that an XML document is valid (maybe it's generated from a database query, for example), you may be able to get away with skipping validation. Depending on how complicated the document rules are, this can save a significant amount of time and memory.

If you have brittle code that gets input data from XML documents, and that code requires that the documents follow a particular DTD or schema, you're probably going to have to validate everything, regardless of how expensive or time-consuming it is.

Parsers that support XML Schema languages

The original XML specification defined the Document Type Definition (DTD) as a way of defining how a valid XML document should look. Most of the DTD syntax came from SGML, and was concerned with validation from a publishing perspective rather than a data typing perspective. In addition, DTDs have a different syntax from XML documents, making it difficult for users to understand the syntax of the document rules.

To address these limitations, the Worldwide Web Consortium (W3C) created the XML Schema language. An XML Schema allows you to define what a valid document looks like with much greater precision. The XML Schema language is very rich, and XML Schema documents can be quite complex. For this reason, XML parsers that support validation with XML Schema tend to be quite large.

Also, be aware that the W3C's XML Schema language isn't the only game in town. Some parsers support other XML schema languages, such as RELAX NG or Schematron (I'll discuss schema languages in a future tutorial.)

The Document Object Model (DOM)

The Document Object Model, or DOM, is an official recommendation of the W3C. It defines an interface that enables programs to access and update the structure of XML documents. When an XML parser claims to support the DOM, that means it implements the interfaces defined in the standard.

At the moment, two levels of the DOM are official recommendations, the cleverly-named DOM Level 1 and DOM Level 2. DOM Level 3 is expected to become official in early 2004. All of the DOM functions discussed in this tutorial are part of DOM Level 1, so the code samples will work with any DOM parser.

What you get from a DOM parser

When you parse an XML document with a DOM parser, you get back *a tree structure* that represents the contents of the XML document. All of the text, elements, and attributes (along with other things I'll discuss shortly) are in the tree structure. The DOM also provides a variety of functions that you can use to examine and manipulate the contents and structure of the tree.

Be aware that a parser *doesn't* keep track of certain things. For example, the amount of whitespace between an attribute and its value isn't stored in the DOM tree. From the parser's perspective, these three attributes are exactly the same:

- `type="common"`
- `type =`
  `"common"`
- `type='common'`

The DOM doesn't give you any way of knowing how the original document was coded. As another example, an XML parser will not tell you if an empty element was coded with one tag or two. (For example, was an XHTML horizontal rule coded as

`<hr/>` or `<hr></hr>`?) The information that an XML parser is required to keep track of is called the **XML Infoset** (see Resources).

The Simple API for XML (SAX)

The Simple API for XML (SAX) API is an alternate way of working with the contents of XML documents. It was designed to have a smaller memory footprint, but it puts more of the work on the programmer. SAX and DOM are complementary; each is appropriate on different occasions.

A *de facto* standard, SAX was originally developed by David Megginson with input from many users across the Internet. See Resources for the complete SAX standard. Your parser's documentation probably describes the SAX standard as well.

What you get from a SAX parser

When you parse an XML document with a SAX parser, the parser generates a series of events as it reads the document. It's up to you to decide what to do with those events. Here's a small sampling of the events you can get when you parse an XML document:

- The `startDocument` event.
- For each element, a `startElement` event at the start of the element, and an `endElement` event at the end of the element.
- If an element contains content, there will be events such as `characters` for additional text, `startElement` and `endElement` for child elements, and so forth.
- The `endDocument` event.

As with DOM, a SAX parser won't reveal certain details like the order in which attributes appeared.

JDOM

Although SAX and DOM provide many useful functions, some tasks are more complicated than developers would like them to be. In keeping with the open source community's history of creating tools wherever a need exists, Java technology experts Jason Hunter and Brett McLaughlin created JDOM, a Java library that greatly simplifies working with XML documents.

Like DOM, JDOM provides a tree of objects that represent the XML document, but the way those objects work is much more intuitive for Java programmers. Keep in mind that JDOM includes adapters for using ordinary SAX or DOM parsers in the background; JDOM provides adapters for all major (and several minor) Java XML parsers, so you don't have to worry about whether your Java XML parser supports JDOM or not. JDOM uses a parser in the background without any intervention from you.

How to choose a parser

I'll talk about this in a lot more detail later, but in general, you should use a DOM parser when:

- You need to know a lot about the structure of a document
- You need to change the structure of a document (maybe you want to sort elements, add some new ones, and so forth)
- You need to refer to the parsed information more than once

Continuing these broad generalizations, you should use a SAX parser when:

- You don't have much memory (your *machine* doesn't have much memory, that is)
- You only need a few elements or attributes from the XML document
- You'll only use the parsed information once

Finally, take a look at the JDOM API. JDOM's memory usage is less than DOM, but not as good as SAX. Also, if you want to do validation (a topic I don't discuss in this tutorial), JDOM requires that you configure the underlying parser; JDOM doesn't do validation itself. That being said, if JDOM does everything you need and it works quickly enough for you, it will simplify the coding you have to do.

Section 3. The Document Object Model (DOM)

A common interface for XML structures

The DOM is an interface for working with the structure of XML documents. A project of the W3C, the DOM was designed to provide a set of objects and methods to make life simpler for programmers. (Technically, the DOM predates XML; the early DOM work was meant for HTML documents.) Ideally, you should be able to write a program that uses one DOM-compliant parser to process an XML document, then switch to another DOM-compliant parser without changing your code.

When you parse an XML document with a DOM parser, you get a hierarchical data structure (a DOM tree) that represents everything the parser found in the XML document. You can then use functions of the DOM to manipulate the tree. You can search for things in the tree, move branches around, add new branches, or delete parts of the tree.

As I mentioned earlier, certain information is not available from the XML parser. Individual parsers might implement functions that let you get more information about the source document, but those functions are not part of the DOM, and aren't likely to work with any other parser. See the XML Infoset standard (in Resources) for more information.

Know your Nodes

You already know that a DOM parser returns a tree structure to you; that DOM tree contains a number of `Node` s. From a Java-language perspective, a `Node` is an interface. The `Node` is the base datatype of the DOM; everything in a DOM tree is a `Node` of one type or another.
DOM Level 1 also defines a number of subinterfaces to the `Node` interface:

- `Element`: Represents an XML element in the source document.
- `Attr`: Represents an attribute of an XML element.
- `Text`: The content of an element. This means that an element with text contains text node children; the text of the element is *not* a property of the element itself.
- `Document`: Represents the entire XML document. Exactly one `Document` object exists for each XML document you parse. Given a `Document` object, you can find the root of the DOM tree; from the root, you can use DOM functions to read and manipulate the tree.

Additional node types are: `Comment`, which represents a comment in an XML file; `ProcessingInstruction`, which represents a processing instruction; and `CDATASection`, which represents a CDATA section. Chances are you won't need these node types, but they're there if you need them.
Note: The words "element" and "tag" have different meanings. An **element** is a starting element, an ending element, and everything in between, including attributes, text, comments, and child elements. A **tag** is a pair of <angle brackets> and everything between them, including the element name and any attributes. For example, `<p class="blue">` is a tag, as is `</p>`; `<p class="blue">The quick brown fox</p>` is an element.

Common DOM methods

When you're working with the DOM, you'll often use the following methods:

- `Document.getDocumentElement()`: Returns the root of the DOM tree. (This function is a method of the `Document` interface; it isn't defined for other subtypes of `Node`.)
- `Node.getFirstChild()` and `Node.getLastChild()`: Return the first or last child of a given `Node`.
- `Node.getNextSibling()` and `Node.getPreviousSibling()`: Return the next or previous sibling of a given `Node`.
- `Element.getAttribute(String attrName)`: For a given `Element`, return the value of the attribute named `attrName`. If you want the value of the `"id"` attribute, use `Element.getAttribute("id")`. If that attribute doesn't exist, the method returns an empty string (`""`).

Your first sample application!

For your first application, I'll show you `DomOne.java`, a simple piece of Java code that does four things:

1. Scans the command line for the name of an XML file
2. Creates a parser object
3. Tells the parser object to parse the XML file named on the command line
4. Walks through the resulting DOM tree and prints the contents of the various nodes to standard output

That's it! Although it's hardly an impressive piece of code, this will illustrate the basic DOM methods and how you invoke them. I'll move on to more difficult feats later. For now, take a look at the source code.

Step 1: Scan the command line

Scanning the command line is relatively straightforward. You simply look for an argument, and assume that argument is a filename or a URI. If no argument is on the command line, you print an error message and exit:

```
 public static void main(String argv[])
{
  if (argv.length == 0 ||
      (argv.length == 1 && argv[0].equals("-help")))
  {
    System.out.println("\nUsage:  java DomOne uri");
    System.out.println("   where uri is the URI of the XML " +
                       "document you want to print.");
    System.out.println("   Sample:  java DomOne sonnet.xml");
    System.out.println("\nParses an XML document, then writes " +
                       "the DOM tree to the console.");
    System.exit(1);
  }

  DomOne d1 = new DomOne();
  d1.parseAndPrint(argv[0]);
}
```

All of the examples in this tutorial have similar code for processing the command line; I won't discuss this code from this point forward.

Step 2: Create a parser object

Assuming the command line is okay, the next task is to create a parser object. First-generation XML parsers required you to instantiate a particular class. To use the Apache XML Project's Xerces parser, you'd instantiate `org.apache.xml.parsers.DOMParser`; to use another parser, you would instantiate a class specific to that parser. The downside of this, of course, is that switching parsers means you have to change your source code. With today's parsers, you can use factory classes that shield your code from knowing what parser class you're using.
Here's how you create the factory object, then have the factory object create the

parser:

```
 public void parseAndPrint(String uri)
{
  Document doc = null;

  try
  {
    DocumentBuilderFactory dbf =
      DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();
    . . .
```

The `DocumentBuilder` object is your parser. It implements all of the DOM methods you need to parse and process the XML file. Notice that you're creating these inside a `try` statement; a number of things can go wrong when you create a parser and parse a document, so you need to catch any errors that might occur. The classes `DocumentBuilderFactory` and `DocumentBuilder` are defined by JAXP. I'll talk about JAXP shortly; if you just can't wait to get the details, see A quick word about JAXP on page .

Step 3: Parse the XML file

Now that you've created the parser, telling it to parse the file (or URI) is simple:

```
 doc = db.parse(uri);
```

That's it! You may remember from Step 2 that `doc` is an instance of the `Document` interface. In other words, parsing the XML file gives you a `Document` structure that you can examine to determine what your XML file contains.

Step 4: Print the contents of the DOM tree

The final task is to print the contents of the DOM tree. Because everything in a DOM tree is a `Node` of one type or another, you'll use a recursive method to go through the tree and print everything in it. The strategy will be to call the method to print a node. The method will print that node, then call itself for each of that node's children. If those children have children, the method will call itself for them as well. Here's a sample of that code:

```
 if (doc != null)
     printDomTree (doc);
  . . .
}

public void printDomTree(Node node)
{
  int type = node.getNodeType();
  switch (type)
  {
    // print the document element
    case Node.DOCUMENT_NODE:
    {
      System.out.println("<?xml version=\"1.0\" ?>");
       printDomTree (((Document)node).getDocumentElement());
```

```
        break;
      }
```

The `printDomTree` method takes a `Node` as its argument. If that `Node` is a document node, you'll print out an XML declaration, then call `printDomTree` for the document element (the XML element that contains the rest of the document). The document element will almost certainly have children, so `printDomTree` will call itself for each of those children as well. This recursive algorithm walks through the entire DOM tree and prints its contents to the command line.
Here's how `printDomTree` handles an element node:

```
    case Node.ELEMENT_NODE:
      {
        System.out.print("<");
        System.out.print(node.getNodeName());

        NamedNodeMap attrs = node.getAttributes();
        for (int i = 0; i < attrs.getLength(); i++)
          printDomTree (attrs.item(i));

        System.out.print(">");

        if (node.hasChildNodes())
        {
          NodeList children = node.getChildNodes();
          for (int i = 0; i < children.getLength(); i++)
            printDomTree (children.item(i));
        }

        System.out.print("</");
        System.out.print(node.getNodeName());
        System.out.print('>');

        break;
      }
```

For an element node, you need to print a left angle bracket and the node name. If this element has attributes, you invoke `printDomTree` for each attribute. (Technically, you could use `Attr.getName()` and `Attr.getValue()` to print the attributes here, but I'm trying to illustrate that everything in the DOM tree is a `Node`.) Once you've printed all the attributes, you invoke `printDomTree` again for each child element this node contains. The final step is to print out the ending element after all of the child elements have been processed.
Processing attribute nodes and text nodes is trivial. For an attribute, you output a space, the name of the attribute, an equals sign and an opening double quote, the value of the attribute, and a closing double quote. (This is a shortcut; it's possible for an attribute's value to contain single or double quotes. Feel free to fix that if you like.)
Processing text nodes is simplest of all; you simply write out the text. Here's the code for these two node types:

```
    case Node.ATTRIBUTE_NODE:
      {
        System.out.print(" " + node.getNodeName() + "=\"" +
                          ((Attr)node).getValue() + "\"");
        break;
      }
      case Node.TEXT_NODE:
      {
        System.out.print(node.getNodeValue());
        break;
      }
```

Although all you're doing here is printing out the contents of the DOM tree, most XML applications built around the DOM use this four-step pattern, with the fourth step being a recursive processing routine.
To see the complete source, visit DomOne.java.

Running DomOne

Now for the moment you've no doubt been waiting for -- running this code. I'm assuming that you've already downloaded and installed the tools you need; if that's not the case, see Setting up your machine.
From the directory where you unzipped the examples, type:

```
javac DomOne.java
```

(Remember that the Java language is case-sensitive.) Assuming there aren't any errors, type:

```
java DomOne sonnet.xml
```

to parse and display the file `sonnet.xml`.
If everything goes as expected, you should see something like this:

```
 C:\xml-prog-java>java DomOne sonnet.xml
<?xml version="1.0" ?>
<sonnet type="Shakespearean">
  <author>
    <lastName>Shakespeare</lastName>
    <firstName>William</firstName>
    <nationality>British</nationality>
    <yearOfBirth>1564</yearOfBirth>
    <yearOfDeath>1616</yearOfDeath>
  </author>
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    . . .
  </lines>
</sonnet>
```

Getting rid of whitespace

The results generated by `DomOne` include lots of whitespace, including whitespace between elements. In many cases, you may not care about this whitespace. For example, the line breaks and spaces used to indent all of the `<line>` elements aren't significant if all you want to do is get the text of the lines of the sonnet.

Keep in mind that a DOM tree is full of objects, so all of the whitespace nodes are actually Java objects. Those objects take memory, they take processing time to create and destroy, and your code has to find them and ignore them. Clearly, if you could tell the parser not to create these objects at all, that would save time and memory, always a worthy goal.

Fortunately, JAXP's `DocumentBuilderFactory` provides a method for ignoring non-significant whitespace. The `setIgnoringElementContentWhitespace(boolean)` method, wordy as it is, takes the whitespace nodes out of the picture. I'll take a look at it next.

DomTwo

The source code for `DomTwo` is identical to the source code for `DomOne` with one exception: the `setIgnoringElementContentWhitespace` method. Here's how the `parseAndPrint` method looks:

```
 public void parseAndPrint(String uri)
{
  Document doc = null;
  try
  {
    DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
 dbf.setIgnoringElementContentWhitespace(true);
    DocumentBuilder db = dbf.newDocumentBuilder();
    doc = db.parse(uri);
    if (doc != null)
      printDomTree(doc);
    . . .
```

Notice that `setIgnoringElementContentWhitespace()` is a method of `DocumentBuilderFactory`; you set the property on the factory, then ask it to create a parser based on the settings you've configured.

If you compile and run `DomTwo`, your results will be shorter, but considerably less readable than before:

```
 C:\xml-prog-java>java DomTwo sonnet.xml
<?xml version="1.0" ?>
<sonnet type="Shakespearean"><author><lastName>Shakespeare
</lastName><firstName>William</firstName><nationality>Brit
ish</nationality><yearOfBirth>1564</yearOfBirth><yearOfDea
th>1616</yearOfDeath></author><title>Sonnet 130</title><li
nes><line>My mistress' eyes are nothing like the sun,</lin
e><line>Coral is far more red than her lips red.</line><lin
e>If snow be white, why then her breasts are dun,</line><l
. . .
```

Although this output isn't easy to read, removing all those whitespace nodes when you don't need them makes life simpler for your code. If you're generating or processing an XML document for a machine, ignoring whitespace is probably the way to go. On the other hand, if you need to format information for a human, the whitespace will probably be important.

To see the source for `DomTwo`, check out DomTwo.java.

The Document node

Sometimes the `Document` node causes confusion. The `Document` node is different from the root of the XML document. For example, our XML sonnet could look like this:

```
 <?xml version="1.0" ?>
<!-- Does this sonnet have a name other than "Sonnet 130"? -->
<sonnet type="Shakespearean">
  <author>
  . . .
  </author>
  <title>Sonnet 130</title>
  <lines>
  . . .
  </lines>
</sonnet>
<!-- The title of Sting's 1987 album "Nothing Like the Sun"
     came from this sonnet. -->
```

In this case the `Document` has three children: The first comment node, the `<sonnet>` element, and the last comment. The document root is the `<sonnet>` element, which is not the same as the `Document` node itself.
Most of the time this distinction won't matter, but keep in mind there is a difference between the two.

A quick word about JAXP

Before I move on to the SAX API, I'll mention a few things about JAXP, the Java API for XML Parsing. This API specifies certain common tasks that the DOM and SAX standards leave out. Specifically, creating parser objects is not defined by the DOM or SAX standards, and DOM does not define turning features of those parsers on and off.
The `javax.xml.parsers` package is the only part of JAXP I'll discuss here. (Other parts of the JAXP specification relate to transforming XML documents, but they're beyond the scope of this tutorial.) The `javax.xml.parsers` package defines four classes:

`DocumentBuilderFactory`
A factory API that allows you to create a DOM parser. By using the factory API, your code doesn't have to know the actual class that implements the DOM. If you find a better DOM parser, you can start using the new parser without modifying or recompiling your code.

`DocumentBuilder`
Defines the API that lets you get a DOM `Document` object from the DOM parser.

`SAXParserFactory`
A factory API that lets you create a SAX parser. As with `DocumentBuilderFactory`, this API shields you from needing to know the name of the class that implements the SAX parser.

`SAXParser`

An API that wraps the SAX `XMLReader` class. When an `XMLReader` reads a file, it generates SAX events.

This section has covered the `DocumentBuilderFactory` and `DocumentBuilder` classes; I'll take a closer look at `SAXParserFactory` and `SAXParser` in the next section.

Section 4. SAX: The Simple API for XML

SAX overview

While the DOM was being defined at the W3C, a number of concerns were raised in discussion groups on the Web. Some of those concerns were:

- The DOM builds an in-memory tree of an XML document. If the document is really large, the DOM tree may require an enormous amount of memory.
- The DOM tree contains many objects that represent the contents of the XML source document. If you only need a couple of things from the document, creating all those objects is wasteful.
- A DOM parser must build the entire DOM tree before your code can work with it. If you're parsing a large XML document, you could experience a significant delay while waiting for the parser to finish.

Led by David Megginson, the SAX API was defined in response to these concerns. The standard was created as a grass-roots effort, and is supported by virtually all XML parsers.

How SAX works

SAX is a **push** API: You create a SAX parser, and the parser tells you (it pushes events to you) when it finds things in the XML document. Specifically, here's how SAX addresses the concerns mentioned in the previous panel:

- SAX doesn't build an in-memory tree of an XML document. A SAX parser sends you events as it finds things in an XML document. It's up to you to decide how (or if) you want to store that data.
- A SAX parser doesn't create any objects. You have the option of creating objects if you want, but that's your decision, not the parser's.
- SAX starts sending you events immediately. You don't have to wait for the parser to finish reading the document.

SAX events

The SAX API defines a number of events. Your job is to write Java code that does something in response to all of those events. Most likely you'll use SAX's helper classes in your application. When you use the helper classes, you write event handlers for the few events you care about, and let the helper class do the rest of the work. If you don't handle an event, the parser discards it, so you don't have to worry about memory usage, unnecessary objects, and other concerns you'd have if you were using a DOM parser.

Keep in mind that SAX events are **stateless**. In other words, you can't look at an individual SAX event and figure out its context. If you need to know that a certain piece of text occurred inside a `<lastName>` element that's inside an `<author>` element, it's up to you to keep track of what elements the parser has found so far. All the SAX event will tell you is, "Here's some text." It's your job to figure out what element that text belongs to.

Some commonly used SAX events

In a SAX-based XML application, you'll spend most of your time looking at five basic events. I'll cover those here, and take a look at error handling events later.

`startDocument()`
Tells you that the parser has found the start of the document. This event doesn't pass you any information, it simply lets you know that the parser is about to begin scanning the document.

`endDocument()`
Tells you that the parser has found the end of the document.

`startElement(...)`
Tells you that the parser has found a start tag. This event tells you the name of the element, the names and values of all of the element's attributes, and gives you some namespace information as well.

`characters(...)`
Tells you that the parser has found some text. You get a character array, along with an offset into the array and a length variable; together, those three variables give you the text the parser found.

`endElement(...)`
Tells you that the parser has found an end tag. This event tells you the name of the element, as well as the associated namespace information.

`startElement()`, `characters()`, and `endElement()` are the most important events; I'll take a closer look at those events next. (`startDocument` and `endDocument` simply tell you when parsing starts and ends.) All of the events here are part of the `ContentHandler` interface; other SAX interfaces are defined to handle errors as well as entities and other rarely-used things.

The startElement() event

The `startElement()` event tells you that the SAX parser has found the start tag for an element. The event has four parameters:

`String uri`
: The namespace URI. In these examples, none of the XML documents use namespaces, so this will be an empty string.

`String localName`
: The unqualified name of the element.

`String qualifiedName`
: The qualified name of the element. This is the namespace prefix combined with the local name of the element.

`org.xml.sax.Attributes attributes`
: An object that contains all of the attributes for this element. This object provides several methods to get the name and value of the attributes, along with the number of attributes the element has.

If your XML application is looking for the contents of a particular element, the `startElement()` event tells you when that element begins.

Note: Processing namespaces requires a namespace-aware parser. By default, the parsers created by the `SAXParserFactory` and `DocumentBuilderFactory` objects are not namespace aware. I'll discuss namespaces in more detail in an upcoming tutorial on advanced XML programming.

The characters() event

The `characters()` event contains the characters that the parser has found in the source file. In the spirit of minimizing memory consumption, this event contains an array of characters; this is more lightweight than a Java `String` object. Here are the parameters for the `characters()` event:

`char[] characters`
: An array of characters found by the parser. The `start` and `length` parameters indicate the portion of the array that generated this event.

`int start`
: The index of the first character in the `characters` array that belongs to this event.

`int length`
: The number of characters in this event.

If your XML application needs to store the contents of a particular element, the `characters()` event handler is where you'll put the code that stores the content.

Note: The SAX standard does not specify the contents of the `characters` array outside the characters related to this event. It might be possible to look in the character array to find details on previous or future events, but that's not advisable.

Even if a given parser supplies those extra characters in the current release, future releases of the parser are free to change the way the characters outside the current event are handled.
The bottom line: It might be tempting to look outside the range specified by `start` and `length`, but *don't*.

The endElement() event

The `endElement()` event tells you that the parser has found the end tag for a particular element. It has three parameters:

`String uri`
The namespace URI.

`String localName`
The unqualified name of the element.

`String qualifiedName`
The qualified name of the element. This is the namespace prefix combined with the local name of the element.

A typical response to this event is to modify state information in your XML application.

Your first SAX application

Now that you've seen the most common SAX events, I'll show you a simple SAX application. This application, `SaxOne.java`, will work similarly to your first DOM application, `DomOne.java`. This SAX application will write the contents of events out to the console, so when you run `java SaxOne sonnet.xml`, the results should be the same results you got earlier from `DomOne`.
You code has three tasks. These are almost identical to the four tasks `DomOne` had:

- Scan the command line for the name (or URI) of an XML file.
- Create a parser object.
- Tell the parser object to parse the XML file named on the command line, and tell it to send your code all of the SAX events it generates.

For `DomOne`, you had four tasks because you processed the DOM tree after parsing was complete; because you're using SAX here, the processing happens during parsing. When parsing is finished, you're through with the SAX events.
I'll examine the three tasks in more detail next.

SAX step 1: Scan the command line

As with `DomOne`, you simply scan the command line for an argument. If there isn't an argument, you print an error message and exit. Otherwise, you assume that the first argument is the name or URI of an XML file:

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

```
{
  if (argv.length == 0 ||
      (argv.length == 1 && argv[0].equals("-help")))
  {
    // Print an error message and exit...
  }
  SaxOne s1 = new SaxOne();
  s1.parseURI(argv[0]);
}
```

SAX step 2: Create a parser object

Your next task is to create a parser object. You'll use JAXP's `SAXParserFactory` API to create a `SAXParser`:

```
 public void parseURI(String uri)
{
  try
  {
 SAXParserFactory spf = SAXParserFactory.newInstance();
    SAXParser sp = spf.newSAXParser();
    . . .
```

That's all you need to do. Your final step is to have the `SAXParser` actually parse the file.
(In case you missed the brief discussion of JAXP, see A quick word about JAXP in the DOM section of this tutorial.)

SAX step 3: Parse the file and handle any events

Now that you've created your parser object, you need to have it parse the file. That's done with the `parse()` method:

```
    . . .

 sp.parse(uri, this);
  }
  catch (Exception e)
  {
    System.err.println(e);
  }
}
```

Notice that the `parse()` method takes two arguments. The first is the URI of the XML document, while the second is an object that implements the SAX event handlers. In the case of `SaxOne`, you're extending the SAX `DefaultHandler` interface:

```
 public class SaxOne
 extends DefaultHandler
```

`DefaultHandler` has an implementation of a number of event handlers. These implementations do nothing, which means all your code has to do is implement

handlers for the events you care about.
Note: The exception handling above is sloppy; as an exercise for the reader, feel free to handle specific exceptions, such as SAXException or java.io.IOException.

Implementing event handlers

When you told your SAXParser to parse the file, you claimed that SaxOne extends the DefaultHandler interface. The final thing I'll show you is how SaxOne implements some of the event handlers in that interface. A major benefit of the DefaultHandler interface is that it shields you from having to implement all of the event handlers. DefaultHandler implements all of the event handlers; you just implement the ones you care about.
First, here's what you do for startDocument():

```
  public void startDocument()
  {
    System.out.println("<?xml version=\"1.0\"?>");
  }
```

This is clearly cheating; you're simply writing out a basic XML declaration, regardless of whether one was in the original XML document or not. Currently the base SAX API doesn't return the details of the XML declaration, although an extension API (defined in SAX2 Version 1.1) has been proposed to give you that information.
Next, here's what you do for startElement():

```
  public void startElement(String namespaceURI, String localName,
                           String rawName, Attributes attrs)
  {
    System.out.print("<");
    System.out.print(rawName);
    if (attrs != null)
    {
      int len = attrs.getLength();
      for (int i = 0; i < len; i++)
      {
        System.out.print(" ");
        System.out.print(attrs.getQName(i));
        System.out.print("=\"");
        System.out.print(attrs.getValue(i));
        System.out.print("\"");
      }
    }
    System.out.print(">");
  }
```

You print the name of the element, then you print all of its attributes. When you're done, you print the ending bracket (>). As I mentioned before, for empty elements (elements such as XHTML's <hr/>), you get a startElement() event followed by an endElement() event. You have no way of knowing whether the source document was coded <hr/> or <hr></hr>.

More event handling

The next important handler is the `characters()` handler:

```
 public void characters(char ch[], int start, int length)
{
  System.out.print(new String(ch, start, length));
}
```

Because all you're doing here is printing the XML document back to the console, you're simply printing the portion of the character array that relates to this event.
The next two handlers I'll show you are `endElement()` and `endDocument()`:

```
 public void endElement(String namespaceURI, String localName,
                        String rawName)
{
  System.out.print("</");
  System.out.print(rawName);
  System.out.print(">");
}

public void endDocument()
{

}
```

When you get the `endElement()` event, you simply write out the end tag. You're not doing anything with `endDocument()`, but it's listed here for completeness. Because the `DefaultHandler` interface ignores `endDocument()` by default, you could leave this method out of your code and get the same results.
Next, I'll take a look at error handling in SAX applications.

Error handling

SAX defines the `ErrorHandler` interface; it's one of the interfaces implemented by `DefaultHandler`. `ErrorHandler` contains three methods: `warning`, `error`, and `fatalError`.
`warning` corresponds to a warning, a condition not defined as an error or a fatal error by the XML specification. `error` is called in response to an error event; as an example, the XML Spec defines breaking a validity rule as an error. I haven't covered validity checking, but a sonnet that doesn't contain 14 `<line>` elements would be an error. Finally, `fatalError` is defined by the XML spec as well; an attribute value that isn't quoted is a fatal error.
The three events defined by `ErrorHandler` are:

`warning()`
Issued in response to a warning as defined by the XML specification.

`error()`
Issued in response to what the XML spec considers an error condition.

`fatalError()`
Issued in response to a fatal error as defined by (you guessed it) the XML spec.

Warnings and errors are typically related to things like entities and DTDs (topics I'm not discussing here), while fatal errors tend to be violations of basic XML syntax (tags that aren't nested properly, for example). Here's the syntax of the three error handlers:

```
 public void warning(SAXParseException ex)
{
  System.err.println("[Warning] "+
                   getLocationString(ex)+": "+
                   ex.getMessage());
}

public void error(SAXParseException ex)
{
  System.err.println("[Error] "+
                   getLocationString(ex)+": "+
                   ex.getMessage());
}

public void fatalError(SAXParseException ex)
  throws SAXException
{
  System.err.println("[Fatal Error] "+
                   getLocationString(ex)+": "+
                   ex.getMessage());
  throw ex;
}
```

The handlers above use a private method named `getLocationString` to give more details about the error. The `SAXParseException` class defines methods such as `getLineNumber()` and `getColumnNumber()` to provide the line and column number where the error occurred. `getLocationString` merely formats this information into a useful string; putting this code into a separate method means you don't have to include this code in every error handler. See SaxOne.java for all the details.

Running SaxOne

Running `SaxOne` is pretty simple. First of all, make sure you've set up the XML parser and code samples as described in Setting up your machine. Next, simply type `java SaxOne sonnet.xml`. You should see something like this:

```
 C:\xml-prog-java>java SaxOne sonnet.xml
<?xml version="1.0" ?>
<sonnet type="Shakespearean">
  <author>
    <lastName>Shakespeare</lastName>
    <firstName>William</firstName>
    <nationality>British</nationality>
    <yearOfBirth>1564</yearOfBirth>
    <yearOfDeath>1616</yearOfDeath>
  </author>
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    . . .
  </lines>
```

```
</sonnet>
```

To see the complete code listing, look at SaxOne.java.

Dealing with whitespace in SAX applications

You probably noticed that running `SaxOne` gave you nicely-formatted XML. That's because you're keeping all of the whitespace nodes that were in the original XML document. `SaxOne` implements the `ignorableWhitespace()` method:

```
 public void ignorableWhitespace(char ch[], int start, int length)
{
  characters(ch, start, length);
}
```

All this method does is call the `characters()` event with the same parameters (`ignorableWhitespace()` and `characters()` have the same signature). If you want to leave out the whitespace, simply remove the `ignorableWhitespace()` handler from your code. Here are the results from the `SaxTwo` application:

```
 C:\xml-prog-java>java SaxTwo sonnet.xml
<?xml version="1.0" ?>
<sonnet type="Shakespearean"><author><lastName>Shakespeare
</lastName><firstName>William</firstName><nationality>Brit
ish</nationality><yearOfBirth>1564</yearOfBirth><yearOfDea
th>1616</yearOfDeath></author><title>Sonnet 130</title><li
nes><line>My mistress' eyes are nothing like the sun,</lin
e><line>Coral is far more red than her lips red.</line><lin
e>If snow be white, why then her breasts are dun,</line><l
. . .
```

If you'd like to see all the source code for `SaxTwo`, check SaxTwo.java.

Section 5. JDOM

JDOM overview

Another response to the perceived complexity of the Document Object Model is JDOM. Created by Brett McLaughlin and Jason Hunter, it uses the 80-20 rule to provide a simple API for 80% of the most common XML processing functions. It doesn't attempt to replace the DOM, and it only works in the Java language at this point.

As you'll see in this section, JDOM makes some common tasks extremely easy. Parsing is simple. Adding items to a parsed XML document is simple. Writing a parsed XML document as an XML file is simple.

An aside on the **80-20 rule**: In the software industry, this means that 80% of the usefulness of your software comes from 20% of your code. JDOM's goal is to write that 20% and provide a simple API for it. The first expression of the 80-20 rule was made by Italian economist Vilfredo Pareto in 1906, when he observed that 80% of the land in Italy was owned by 20% of the people. The *Pareto principle* (aka the 80-20 rule) was conceptualized in the 1930s by quality guru Dr. Joseph Juran; in the following years, scientists in many other disciplines noticed that 20% of something was often responsible for 80% of the results. If you'd like to impress your friends with more trivia about the 80-20 rule, visit http://www.juran.com/search_db.cfm and search for "Pareto."

Goals of JDOM

To quote from the JDOM Web site (see Resources), "There is no compelling reason for a Java API to manipulate XML to be complex, tricky, unintuitive, or a pain in the neck." Using JDOM, XML documents work like Java objects and use Java collections. I'll show you two JDOM applications similar to `DomOne` and `DomTwo`; you'll notice that the applications are much smaller.

It's important to note that JDOM integrates with both DOM and SAX. You can use DOM trees or SAX events to build JDOM documents. Once you have a JDOM document, you can convert it to a DOM tree or to SAX events easily.

JdomOne

As you'd expect, the code for `JdomOne` is structurally similar to `DomOne` and `SaxOne`. Fortunately, the code you have to write is much smaller. First you check the command line for an XML file name, then you have the logic of the program:

```
 try
{
  SAXBuilder sb = new SAXBuilder();
  Document doc = sb.build(new File(argv[0]));
  XMLOutputter xo = new XMLOutputter();
  xo.output(doc, System.out);
}
catch (Exception e)
{
  e.printStackTrace();
}
```

That's it! Notice that JDOM defines an `XMLOutputter` class that writes the XML file out to the command line. Outputting a parsed document as XML source is a common task, so JDOM provides a class to do that for you. For the first two lines of code here, both `SAXBuilder` and `Document` are part of the JDOM package (`org.jdom.input.SAXBuilder` and `org.jdom.Document`, respectively).

If you're combining JDOM and DOM, be aware that some classes (such as `Document`) are defined by both packages. You have to fully qualify those class names so that the Java compiler knows which `Document` you're talking about.

Running JdomOne

Before you run a JDOM application, you need to set up your machine as described in Setting up your machine. Running `JdomOne` gives you the same results as your earlier DOM and SAX applications:

```
 C:\xml-prog-java>java JdomOne sonnet.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sonnet SYSTEM "sonnet.dtd">
<sonnet type="Shakespearean">
  <author>
    <lastName>Shakespeare</lastName>
    <firstName>William</firstName>
    <nationality>British</nationality>
    <yearOfBirth>1564</yearOfBirth>
    <yearOfDeath>1616</yearOfDeath>
  </author>
  <title>Sonnet 130</title>
  <lines>
    <line>My mistress' eyes are nothing like the sun,</line>
    <line>Coral is far more red than her lips red.</line>
    <line>If snow be white, why then her breasts are dun,</line>
    <line>If hairs be wires, black wires grow on her head.</line>
    . . .
  </lines>
</sonnet>
```

Notice that JDOM added the `encoding` attribute to the XML declaration and added the `DOCTYPE` declaration. The complete source code is in JdomOne.java.

Removing whitespace

To provide a JDOM version of `DomTwo`, you need to remove all of the ignorable whitespace. As you'd expect, JDOM makes it easy to do this. Here's how the code looks:

```
 SAXBuilder sb = new SAXBuilder();
Document doc = sb.build(new File(argv[0]));
XMLOutputter xo = new XMLOutputter();
 xo.setTrimAllWhite(true);
xo.output(doc, System.out);
```

All you have to do is add the `setTrimAllWhite()` method to your code. You tell the `XMLOutputter` to remove all of the whitespace, and get the results you'd expect.
For the complete sample, see JdomTwo.java.

The joys of JDOM

At this point, you should have a good idea of the simplicity of JDOM. You should definitely take a look at this API; if it does everything you need, it can greatly simplify your development process. Best of all, JDOM has adapters for all the common XML parsers (Xerces from the Apache XML Project, Oracle's XML parser, Crimson from Sun, an adapter for the JAXP APIs, and an adapter for the IBM's XML4J parser). Although it's beyond the scope of this tutorial, you can also create your own adapter to make other data sources look like XML documents. For example, you could have information from a relational database or an LDAP directory appear to be an ordinary DOM tree. You could then use JDOM methods to manipulate that data any way you want.

Section 6. Summary

Summary

This tutorial has shown you several different parsers and interfaces for working with XML documents in Java technology. The interfaces are portable across different parser implementations, meaning you can change the infrastructure behind your XML application without changing your source code. The tutorial also discussed the strengths and weaknesses of the various APIs, and looked at some scenarios for using XML and the various APIs to solve real-world problems.
I hope this gets you started writing your own XML applications in the Java language.

What wasn't covered

To keep this tutorial at a manageable length, I didn't cover a number of advanced topics. I will cover all of these in an advanced XML programming tutorial here at *developerWorks*. Some of the topics to be covered are:

- Handling namespaces
- Validation with DTDs and XML schema languages
- Converting between APIs (generating DOM trees from SAX events, for example)
- Creating DOM and JDOM objects without an XML source document
- Stopping a SAX parser before it finishes reading the XML source
- Moving nodes around in a DOM tree
- DOM serialization functions
- Advanced DOM topics (ranges, traversal, and other parts of the Level 2 and Level 3 specs)

Resources

Learn

- **Tutorials**
  - "Introduction to XML" covers the basics of XML. A good place to start if you're new to markup languages (*developerWorks*, August 2002).
  - "Understanding DOM" covers the Document Object Model in far more depth than I've covered here (*developerWorks*, July 2003).
  - "Understanding SAX" gives an in-depth overview of the Simple API for XML (*developerWorks*, July 2003).

- **Books**
  - *Learning XML, 2nd Edition*, by Erik T. Ray, published by O'Reilly, is a very good (and very popular) book for XML beginners (http://www.oreilly.com/catalog/learnxml2/).
  - *XML in a Nutshell, 2nd Edition*, by Elliotte Rusty Harold and W. Scott Means, published by O'Reilly, is a great reference to keep on your desk, especially if you know the basics of XML (http://www.oreilly.com/catalog/xmlnut2/).
  - *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*, by Elliotte Rusty Harold (published by Addison-Wesley) is an in-depth look at all of the APIs I've covered here (http://www.awprofessional.com). The author is one of the XML community's leading authors. (The Addison-Wesley Web site doesn't have a direct link to the book; search for "Processing XML" to find the book's Web page.)
  - *Java and XML, 2nd Edition*, by Brett McLaughlin, is another excellent discussion of using the APIs covered here with Java technology (http://www.oreilly.com/catalog/javaxml2/).
  - O'Reilly's SAX2 is the best source I've seen for in-depth information about the SAX API (http://www.oreilly.com/catalog/sax2/). It's written by David Brownell, the maintainer of SAX, so buying the book helps David continue his work.

- **Specs**
  - The DOM Level 1 spec is available at the W3C's Web site (http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/). All the specs for DOM Levels 1, 2, and 3 are available on the DOM Technical Reports page at the W3C (http://www.w3.org/DOM/DOMTR).
  - A formal JDOM spec has not been written at this point (January 2004); the JDOM Web site states that the JDOM Javadoc files are the spec for now (http://www.jdom.org/docs/apidocs/). In addition, the JDOM documentation

    page has a list of useful JDOM articles (http://www.jdom.org/downloads/docs.html).
  - All things SAX, including a quick-start document, a discussion of the APIs, and the SAX Javadoc files, are at www.saxproject.org.
  - I mentioned XML Infoset briefly in the Introduction section; you can see the Infoset spec in all its glory at the W3C's Web site (http://www.w3.org/TR/xml-infoset/).

- **Additional resources**
  - Check out Brett McLaughlin's "XML and Java Technology" forum for more information on how these two technologies intersect.
  - Find a broad array of articles, columns, tutorials, and tips on these two popular technologies at the *developerWorks* Java technology and XML content areas.
  - Find out how you can become an IBM Certified Developer in XML and related technologies.

About the author

Doug Tidwell

(This biography was written by Lily Castle Tidwell, the author's daughter.)

My daddy's name is Doug Tidwell. He has black hair. Daddy wears glasses. He is six feet tall. His eyes are brown.

Daddy likes to play catch with me. He likes to run. He's really fast. He is a really good bike rider. His bike is blue.

My daddy's job is to travel. He goes to China a lot. He has hiked the Great Wall of China. He got a certificate.

I LOVE YOU DADDY!

P.S. Every day that has a **d** in it, you are the goofiest.

You can contact the author, Doug Tidwell, at dtidwell@us.ibm.com.