XML PARSING with DOM and XERCES (part one)

By icarus

Table of Contents

The Big Picture

If you're at all familiar with XML programming, you'll be aware that there are two basic approaches to parsing an XML document. The Simple API for XML (SAX) is one; it parses an XML document in a sequential manner, generating and throwing events for the application layer to process as it encounters different XML elements. This sequential approach enables rapid parsing of XML data, especially in the case of long or complex XML documents; however, the downside is that a SAX parser cannot be used to access XML document nodes in a random or non−sequential manner.

Hence the Document Object Model (DOM). This alternative approach involves building a tree representation of the XML document in memory, and then using built−in methods to navigate through this tree. Once a particular node has been reached, built−in properties can be used to obtain the value of the node, and use it within the script. This tree−based paradigm does away with the problems inherent in SAX's sequential approach, allowing for immediate random access to any node or collection of nodes in the tree.

DOM parsers are available for a variety of different platforms − you can get them for Perl, PHP, Python and C. However, the one that's going to occupy us for the next ten minutes is the Java−based Xerces parser, which includes a very capable DOM parser. Over the next few pages, I'm going to be demonstrating how it works, using some real−world examples to bring home its capabilities and to illustrate how the combination of Java, XML and JSP can be used to develop XML−based Web applications.

Let's get started!

Float Like A Butterfly...

The Xerces Java Parser (version 1.4.4) has been developed by the same people who created Apache, the Web server that is standard on most UNIX systems. Though it's named after a butterfly, this parser is anything but lightweight; it supports the latest version of the DOM standard (Level 2) in addition to the SAX 1.0 standard and the newer SAX 2.0 specification. Note, however, that since XML standards are constantly evolving, using Xerces can sometimes produce unexpected results; take a look at the documentation provided with the parser, and at the information available on its official Web site, for errata and updates.

With the introductions out of the way, let's put together the tools you'll need to get started with Xerces. Here's a quick list of the software you'll need:

1. The Java Development Kit (JDK), available from the Sun Microsystems Web site (http://java.sun.com)

2. The Apache Web server, available from the Apache Software Foundation's Web site (http://httpd.apache.org)

3. The Tomcat Application Server, available from the Apache Software Foundation's Web site (http://httpd.apache.org)

4. The Xerces parser, available from the Apache XML Project's Web site (http://xml.apache.org)

5. The mod_jk extension for Apache−Tomcat communication, available from the Jakarta Project's Web site (http://httpd.apache.org)

Installation instructions for all these packages are available in their respective source archives. In case you get stuck, you might want to look at http://www.devshed.com/Server_Side/Java/JSPDev, or at the Tomcat User Guide at http://jakarta.apache.org/tomcat/tomcat−3.3−doc/tomcat−ug.html

Nailguns, Going Cheap

I'll begin with something simple. Consider the following XML file, an XML−encoded inventory statement for a business selling equipment to Quake enthusiasts.

```
<?xml version="1.0"?>
<inventory>
<item>
<id>758</id>
<name>Rusty, jagged nails for nailgun</name>
<supplier>NailBarn, Inc.</supplier>
<cost>2.99</cost>
<quantity>10000</quantity>
</item>
<item>
<id>6273</id>
<name>Power pack for death ray</name>
<supplier>QuakePower.domain.com</supplier>
<cost>9.99</cost>
<quantity>10</quantity>
</item>
</inventory>
```

The Xerces DOM parser is designed to read an XML file, build a tree to represent the structures found within it, and expose object methods and properties to manipulate them. This next example demonstrates how, building a simple Java application that initializes the parser and reads the XML file.

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;
import java.io.*;

public class MyFirstDomApp {

// constructor
public MyFirstDomApp (String xmlFile) {

// create a DOM parser
DOMParser parser = new DOMParser();

// parse the document
try {
parser.parse(xmlFile);
Document document = parser.getDocument();
NodeDetails(document);
} catch (IOException e) {
```

```
System.err.println (e);
}
}


// this function prints out information on a specific node
// in this example, the "#document" node
// it then goes to the next node
// and does the same for that
private void NodeDetails (Node node) {
System.out.println ("Node Type:" + node.getNodeType() +
"\nNode
Name:" + node.getNodeName());
if(node.hasChildNodes()) {
System.out.println ("Child Node Type:" +
node.getFirstChild().getNodeType()
+ "\nNode Name:" + node.getFirstChild().getNodeName());
}
}

// the main method to create an instance of our DOM
application
public static void main (String[] args) {
MyFirstDomApp MyFirstDomApp = new MyFirstDomApp (args[0]);
}
}
```

I'll explain what all this gobbledygook means shortly − but first, let's compile and run the code.

```
$ javac MyFirstDomApp.java
```

Assuming that all goes well, you should now have a class file named "MyFirstDomApp.class". Copy this class file to your Java CLASSPATH, and then execute it, with the name of the XML file as argument.

```
$ java MyFirstDomApp /home/me/dom/inventory.xml
```

Here's what the output looks like:

```
Node Type:9
Node Name:#document
Child Node Type:1
Node Name:inventory
```

Now, this might not look like much, but it demonstrates the basic concept of the DOM, and builds the foundation for more complex code. Let's look at the code in detail:

1. The first step is to import all the classes required to execute the application. First come the classes for the Xerces DOM parser, followed by the classes for exception handling and file I/O.

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;
import java.io.*;
```

2. Next, a constructor is defined for the class (in case you didn't already know, a constructor is a method that is invoked automatically when you create an instance of the class).

```
// constructor
public MyFirstDomApp (String xmlFile) {

// create a DOM parser
DOMParser parser = new DOMParser();

// parse the document
try {
parser.parse(xmlFile);
Document document = parser.getDocument();
NodeDetails(document);
} catch (IOException e) {
System.err.println (e);
}
}
```

As you can see, the constructor uses the parse() method to perform the actual parsing of the XML document; it accepts the XML file name as method argument. This method call is enclosed within a "try−catch" error handling block, in order to gracefully recover from errors.

The end result of this parsing is a DOM tree consisting of a single root and its child nodes, each of which exposes methods that describe the object in greater detail.

3. The getDocument() method returns an object representing the entire XML document; this object reference is then passed on to the NodeDetails() method to display information about itself, and its children.

```
// this function prints out information on a specific node
// in this example, the "#document" node
// it then goes to the next node
// and does the same for that
private void NodeDetails (Node node) {
```

```
System.out.println ("Node Type:" + node.getNodeType() +
"\nNode Name:" +
node.getNodeName());
if(node.hasChildNodes()) {
System.out.println ("Child Node Type:" +
node.getFirstChild().getNodeType() + "\nNode Name:" +
node.getFirstChild().getNodeName());
}
}
```

4. Once a reference to a node has been obtained, a number of other methods and properties become available to obtain the name and value of that node, as well as references to parent and child nodes. In the code snippet above, I've used the getNodeType() and getNodeName() methods of the Node object to obtain the node type and name respectively. Similarly, the hasChildNodes() method can be used to find out if a node has child nodes under it, while the getFirstChild() method can be used to get a reference to the first child node.

In case you're wondering about the getNodeType() method − every node is of a specific type, and this method returns a numeric and string constant corresponding to the node type. Here's the list of available types:

Type Type Description Name (num) (str)
−−
1 ELEMENT_NODE Element The element name
2 ATTRIBUTE_NODE Attribute The attribute name
3 TEXT_NODE Text #text
4 CDATA_SECTION_NODE CDATA #cdata−section
5 ENTITY_REFERENCE_NODE Entity reference The entity reference name
6 ENTITY_NODE Entity The entity name
7 PROCESSING_INSTRUCTION_NODE PI The PI target
8 COMMENT_NODE Comment #comment
9 DOCUMENT_NODE Document #document
10 DOCUMENT_TYPE_NODE DocType Root element
11 DOCUMENT_FRAGMENT_NODE DocumentFragment #document−fragment
12 NOTATION_NODE Notation The notation name

Delving Deeper

As you must have figured out by now, using the DOM parser is fairly easy − essentially, it involves creating a "tree" of the elements in the XML document, and traversing that tree with built−in methods. In the introductory example, I ventured as far as the document element; in this next one, I'll go much further, demonstrating how the parser's built−in methods can be used to navigate to any point in the document tree.

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;
import java.io.*;

public class MySecondDomApp {

// constructor
public MySecondDomApp (String xmlFile) {

// create a Xerces DOM parser
DOMParser parser = new DOMParser();

// parse the document and
// access the root node with its children
try {
parser.parse(xmlFile);
Document document = parser.getDocument();
NodeDetails(document);

} catch (IOException e) {
System.err.println (e);
}
}


// this function drills deeper into the DOM tree
private void NodeDetails (Node node) {

// get the node name
System.out.println (node.getNodeName());

// check if the node has children
if(node.hasChildNodes()) {

// get the child nodes, if they exist
NodeList children = node.getChildNodes();
if (children != null) {
for (int i=0; i< children.getLength(); i++) {

// repeat the process for each child node
```

```
// get the node name
System.out.println ("\t" +
children.item(i).getNodeName());

// check if the node has children
if(children.item(i).hasChildNodes()) {

// get the children, if they exist
NodeList childrenOfchildren =
children.item(i).getChildNodes();
if (childrenOfchildren != null) {

// get the node name
for (int j=0; j< childrenOfchildren.getLength();
j++) {
System.out.println ("\t\t" +
childrenOfchildren.item(j).getNodeName());
}
}
}
}
}
}
}

// the main method to create an instance of our DOM
application
public static void main (String[] args) {
MySecondDomApp MySecondDomApp = new MySecondDomApp (args[0]);
}
}
```

Here's the output:

```
#document
inventory
#text
item
#text
item
#text
```

As demonstrated in the first example, the fundamentals remain unchanged − initialize the parser, read an XML document, get a reference to the root of the tree and start traversing the tree. Consequently, most of the code here remains the same as that used in the introductory example, with the changes occurring only in the NodeDetails() function. Let's take a closer look at this function:

```
// this function drills deeper into the DOM tree
private void NodeDetails (Node node) {

// get the node name
System.out.println (node.getNodeName());

// check if the node has children
if(node.hasChildNodes()) {

// get the child nodes, if they exist
NodeList children = node.getChildNodes();
if (children != null) {
for (int i=0; i< children.getLength(); i++) {

// repeat the process for each child node
// get the node name
System.out.println ("\t" + children.item(i).getNodeName());

// check if the node has children
if(children.item(i).hasChildNodes()) {

// get the children, if they exist
NodeList childrenOfchildren =
children.item(i).getChildNodes();
if (childrenOfchildren != null) {

// get the node name
for (int j=0; j< childrenOfchildren.getLength();
j++) {
System.out.println ("\t\t" +
childrenOfchildren.item(j).getNodeName());
}
}
}
}
}
}
}
```

Once a reference to the root of the tree has been obtained and passed to NodeDetails(), the getChildNodes() function is used to obtain a list of the children of that node. This list is returned as a new NodeList object, which comes with its own methods for accessing individual elements of the node list.

As you can see, one of these methods is the getLength() method, used to obtain the number of child nodes, in order to iterate through them. Individual elements of the node list can be accessed via the item() method, which returns a Node object, which puts us back on familiar territory − the Node object's standard

getNodeName() and getNodeType() methods can now be used to access detailed information about the node.

The process is then repeated for each of these Node objects − a check for further children, a retrieved NodeList, a loop iterating through the child Nodes − until the end of the document tree is reached.

When Laziness Is A Virtue

In the example above, I've manually written code to handle each level of the tree for illustrative purposes − however, in a production environment, doing this is pure insanity, especially since an XML document can have any number of nested levels. A far more professional approach would be to write a recursive function to automatically iterate through the document tree − this results in cleaner, more readable code, and it's also much, much easier to maintain.

In order to understand the difference, consider this next example, which uses a recursive function to parse a more complex XML document. Here's the XML,

```
<?xml version="1.0"?>
<inventory>
<!-- time to lock and load -->
<item>
<id>758</id>
<name>Rusty, jagged nails for nailgun</name>
<supplier>NailBarn, Inc.</supplier>
<cost currency="USD">2.99</cost>
<quantity alert="500">10000</quantity>
</item>
<item>
<id>6273</id>
<name>Power pack for death ray</name>
<supplier>QuakePower.domain.com</supplier>
<cost currency="USD">9.99</cost>
<quantity alert="20">10</quantity>
</item>
</inventory>
```

and here's the Java code to parse it:

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;
import java.io.*;

public class MyThirdDomApp {

// a counter for keeping track of the "tabs"
private int TabCounter = 0;

// constructor
public MyThirdDomApp (String xmlFile) {

// create a Xerces DOM parser
```

```
DOMParser parser = new DOMParser();

// parse the document and
// access the root node with its children
try {
parser.parse(xmlFile);
Document document = parser.getDocument();
NodeDetails(document);
} catch (IOException e) {
System.err.println (e);
}
}

// this is a recursive function to traverse the document tree
private void NodeDetails (Node node) {
String Content = "";
int type = node.getNodeType();

// check if element
if (type == Node.ELEMENT_NODE) {
FormatTree(TabCounter);
System.out.println ("Element: " + node.getNodeName() );

// check if the element has any attributes
if(node.hasAttributes()) {

// if it does, store it in a NamedNodeMap object
NamedNodeMap AttributesList = node.getAttributes();
// iterate through the NamedNodeMap and get the attribute
names and
values
for(int j = 0; j < AttributesList.getLength(); j++) {
FormatTree(TabCounter);
System.out.println("Attribute: " +
AttributesList.item(j).getNodeName() + " = " +
AttributesList.item(j).getNodeValue());
}
}
} else if (type == Node.TEXT_NODE) {

// check if text node and print value
Content = node.getNodeValue();

if (!Content.trim().equals("")){
FormatTree(TabCounter);
System.out.println ("Character data: " + Content);
}
} else if (type == Node.COMMENT_NODE) {
// check if comment node and print value
```

```
Content = node.getNodeValue();
if (!Content.trim().equals("")){
FormatTree(TabCounter);
System.out.println ("Comment: " + Content);
}
}

// check if current node has any children
NodeList children = node.getChildNodes();
if (children != null) {
// if it does, iterate through the collection
for (int i=0; i< children.getLength(); i++) {
TabCounter++;
// recursively call function to proceed to next level
NodeDetails(children.item(i));
TabCounter--;
}
}
}

// this formats the output for the generated tree
private void FormatTree (int TabCounter) {
for(int j = 1; j < TabCounter; j++) {
System.out.print("\t");
}
}

// the main method to create an instance of our DOM
application
public static void main (String[] args) {
MyThirdDomApp MyThirdDomApp = new MyThirdDomApp (args[0]);
}
}
```

Here's the output:

```
Element: inventory
Comment: time to lock and load
Element: item
Element: id
Character data: 758
Element: name
Character data: Rusty, jagged nails for nailgun
Element: supplier
Character data: NailBarn, Inc.
Element: cost
Attribute: currency = USD
```

```
Character data: 2.99
Element: quantity
Attribute: alert = 500
Character data: 10000
Element: item
Element: id
Character data: 6273
Element: name
Character data: Power pack for death ray
Element: supplier
Character data: QuakePower.domain.com
Element: cost
Attribute: currency = USD
Character data: 9.99
Element: quantity
Attribute: alert = 20
Character data: 10
```

Now, wasn't that easier than manually writing code for each level of the document tree?

This should be easily understandable if you're familiar with the concept of recursion. Most of the work happens in the NodeDetails() function, which now includes additional code to iterate through the different levels of the document tree automatically, and to make intelligent decisions about what to do with each node type found.

```
// this is a recursive function to traverse the document tree
private void NodeDetails (Node node) {

// snip

// check if element
if (type == Node.ELEMENT_NODE) {
FormatTree(TabCounter);
System.out.println ("Element: " + node.getNodeName() );

// check if the element has any attributes
if(node.hasAttributes()) {

// if it does, store it in a NamedNodeMap object
NamedNodeMap AttributesList = node.getAttributes();
// iterate through the NamedNodeMap and get the attribute
names
and
values
for(int j = 0; j < AttributesList.getLength(); j++) {
FormatTree(TabCounter);
System.out.println("Attribute: " +
```

```
AttributesList.item(j).getNodeName()
+ " = " + AttributesList.item(j).getNodeValue());
}
}
}

// snip

}

// snip
}
```

If the node is an element, the element name is printed to the standard output device. A check is then performed for element attributes; if they exist, they are returned as a NamedNodeMap object (essentially, an array whose elements can be accessed either by integer or string) and can be processed and displayed using methods exposed by that object. If you know the name of the attribute, the getNamedItem() method can be used to retrieve the corresponding value; if you don't (as in the example above), the getLength() and item() methods can be used in combination with a loop to iterate through the list of attributes.

```
// this is a recursive function to traverse the document tree
private void NodeDetails (Node node) {

// snip

// check if element
if (type == Node.ELEMENT_NODE) {
// snip
} else if (type == Node.TEXT_NODE) {

// check if text node and print value
Content = node.getNodeValue();

if (!Content.trim().equals("")){
FormatTree(TabCounter);
System.out.println ("Character data: " + Content);
}
} else if (type == Node.COMMENT_NODE) {

// check if comment node and print value
Content = node.getNodeValue();
if (!Content.trim().equals("")){
FormatTree(TabCounter);
System.out.println ("Comment: " + Content);
}
}
```

```
// snip
}
```

In a similar manner, it's also possible to check for text nodes, comments and any other node type, and write code to process each type individually. The example above handles text nodes and comments, printing each one to the standard output device as they are encountered. Note that, again, the getNodeValue() function is used to extract the raw value of the node − it must be nice to be so popular!

Finally, once the node has been processed, it's time to see if it has any children, and proceed to the next level of the tree if so.

```
// this is a recursive function to traverse the document tree
private void NodeDetails (Node node) {

// snip

// check if current node has any children
NodeList children = node.getChildNodes();
if (children != null) {
// if it does, iterate through the collection
for (int i=0; i< children.getLength(); i++) {
TabCounter++;
// recursively call function to proceed to next level
NodeDetails(children.item(i));
TabCounter−−;
}
}
}
```

In the event that the node does have children, the children are stored in a NodeList, and the NodeDetails() function is recursively called for each of these nodes. And so on, and so on, ad infinitum...or at least until the entire tree has been processed.

Finally, the very simple FormatTree() method checks the value of the tab counter to determine the current depth within the XML tree, and displays that many spaces in the output in a primitive attempt to represent the data as a tree.

```
// this formats the output for the generated tree
private void FormatTree (int TabCounter) {
for(int j = 1; j < TabCounter; j++) {
System.out.print("\t");
}
}
```

As with most things − easy when you know how.

Obviously, this is just one illustration of the applications of the Xerces DOM parser. This is probably enough to get you started with simple Java/XML applications...but you can do a lot more with Xerces than just this.

In the second part of this article, I'll build on everything you just learnt to demonstrate how the Xerces DOM parser can be combined with JSP to format XML documents for a Web browser. I'll also take a look at the error−handling functions built into the parser, demonstrating how they can be used to trap and catch errors in XML processing. Make sure you come back for that one!

Note: All examples in this article have been tested with JDK 1.3.0, Apache 1.3.11, mod_jk 1.1.0, Xerces 1.4.4 and Tomcat 3.3. Examples are illustrative only, and are not meant for a production environment. YMMV!