Using the IBM XML Parser (XML4J)
Find & Replace Elements in an XML Document v.2

by LindaMay Patterson
PartnerWorld for Developers, AS/400
January 2000

Table of Contents

Introduction

eXtensible Markup Language (XML) has opened the door to the sharing of business information in various ways. Some of the key opportunities to use XML include:

1. to supplement the shortcomings of HTML — XML provides self describing data
2. in business-to-business transactions — similar to how EDI is used today
3. as the common communication vehicle between disparate applications.

HTML has been very successful at providing information to the Web; however, HTML has some limitations. XML provides additional flexibility because the documents are self describing. An interesting feature of using XML for web content is in the area of searches. Search engines are able to use both the tag and the data as part of the search criteria, which allows more precise matches of web content. XML documents can be viewed in XML-enabled browsers by using eXtensible Stylesheet Language (XSL) stylesheets to format the document. Using XML in business-to-business transactions and as the common communication vehicle between disparate applications requires a mechanism that will read and interpret the XML document into a computer friendly form. Application programs require a means to access the individual pieces of information (elements) contained within each XML document. This is accomplished by using the XML parser to render the document in a structured form (hierarchical tree structure) which allows each element of the document to be accessed and manipulated.

This tutorial introduces an XML Document and its Document Type Definition (DTD) and how to access the information contained within an XML document through the use of the IBM XML4J parser.

The Basics

Each XML Document consists of elements specific to that document. Figure 1 shows the structure of as XML element. An element with content has a start tag and an end tag, with the content in between the two tags. Elements without content, often used for structuring, can have a start tag with a slash (/) before the greater than sign (>) to denote that no content exists. Elements can be organized into a structure, much like files are today; the nesting is reflected by the position of the start and end tags.

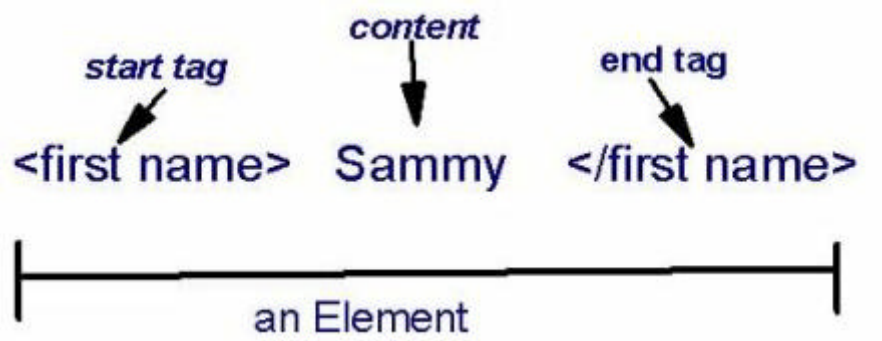

Figure 1: XML Element structure

Any element can have attributes which are used to further define the element. Attributes are defined within the start tag of the element as shown in figure 2.

<first name sex="male"> Sammy </first name>

Figure 2: XML Element with Attribute

Each attribute is a name value pair and the value must be enclosed in double quotation marks ("). Each element may have as many attributes as the designer determines necessary. Each designer must decide if the associated information should be represented as sub elements or attributes — there are no hard and fast rules in this area.

Figure 3 shows the document being used in this tutorial. The document contains data for a catalog. The elements have been indented to easily show the hierarchy of the document.

```
<?XML VERSION="1.0"?>
<catalog season="fall">
        <name>Wally's Fall Outdoors Apparel</name>
        <item>
                <itemname>Wool Jacket</itemname>
                <type>
                        <typename>Male</typename>
                        <cost>$50.00</cost>
                        <description>Lite weight Wool Jacket</description>
                        <number>490195M</number>
                        <weight unit="pound">1.5</weight>
                        <shippingcost />
                </type>
                <type>
                        <typename>Female</typename>
                        <cost>$57.50</cost>
                        <description>Unlined Lite weight Wool Jacket</description>
                        <number>490394W</number>
                        <weight unit="pound">1.2</weight>
                        <shippingcost />
                </type>
        </item>
</catalog>
```

Figure 3 - XML Document for Electronic Catalog

The XML document contains information about two wool jackets, a male version and a female version. Within each <type> </type> tag set is the information for each item (jacket). Notice that the shipping cost tag does not have a end tag but rather a back slash (/) after the term shipping cost within the start tag. This is known as an empty tag which has no content and can be used as a place holder for later use.

A Document Type Definition (DTD) should accompany an XML document to be considered valid. The DTD contains the structure of the XML document and any rules about the relationship between elements and any rules particular to an element. The DTD expresses the hierarchy and the nesting of elements within the structure.

The DTD that defines the structure of the catalog document is defined in Figure 4. This DTD consists of various nested structures with the catalog as the root item. Before the content of this DTD is explained, it is important to understand the following basic principles:

- An element is defined in this form
- An element may contain other sub elements which are enclosed in parenthesis () and listed after the element name.
- Each sub element occurs in the structure according to the associated occurrence rule which is represented by the special character after the element name. The rules are
    - no special character — must occur once and only once
    - asterisk (*) — occurs zero or more times
    - question mark (?) — may occur zero or one time
    - plus (+) — must occur at least one time or more.

The catalog DTD defined in Figure 4 shows the use of some of the occurrence rules. For example, name will occur only once because it has no special character after it. While item can occur none or several times.

```
<!DOCTYPE catalog [
<!ELEMENT catalog (name,item*) >
  <!ELEMENT name (#PCDATA) >
    <!ATTLIST catalog season (winter|spring|summer|fall) #REQUIRED>
  <!ELEMENT item (itemname,type*) >
    <!ELEMENT itemname (#PCDATA) >
    <!ELEMENT type (typename,cost,description,number,
                weight,shippingcost) >
       <!ELEMENT typename (#PCDATA) >      <!ELEMENT cost  (#PCDATA) >
       <!ELEMENT description (#PCDATA) >
       <!ELEMENT number (#PCDATA) >
       <!ELEMENT weight (#PCDATA) >
            <!ATTLIST weight unit (pound|kilogram|gram|ton) #REQUIRED>
       <!ELEMENT shippingcost (#PCDATA) >
]>
```

Figure 4 - Document Type Definition

Some elements contain attributes defined in an attribute list statement. As stated earlier, an attribute is used to add meaning to a particular tag and must be defined within the DTD. For example, the catalog has a season attribute which can have the value of winter, spring, summer or fall. In this particular case, a value is required because of the #REQUIRED keyword. Each element of the document is defined separately for its data type. The #PCDATA (meaning Parsed Character Data) indicating the character (text) data. Currently, XML documents only consist of character data.

Only the basic rules and information to work with this document have been included in this paper. There are various books available that include all the rules and considerations for creating a DTD.

Using the XML Parser

IBM provides the XMLJ4 parser in various products including the AS/400 Toolbox for Java and even on the IBM alphaworks site. The parser is written in Java; therefore, it is portable to operating systems with a Java Virtual Machine (JVM). The parser uses both the DTD and the XML document to create a Document Object Model (DOM) tree which presents the document hierarchically. The DOM provides a group of APIs which allow access to the elements within the tree. Using the DOM APIs, any element within the XML document can be accessed, changed, deleted or added.

The XML parser uses the DTD to validate the document which involves ensuring the XML document follows all of the rules specified in the DTD. For example, the DTD rules can specify the valid set of tags, the valid element nesting rules and the attributes which are associated with a particular element. The XML parser also uses the DTD to help in the creation of the DOM for a particular XML document. The DOM is a representation of the document used by applications at runtime to query and update information within an XML document.

Figure 5 contains an example of the DOM tree for this document. It does not represent the content within the tree structure, the tree must be visualized as a grove or forest of trees representing that content rather than this single structure. Each rectangle in Figure 5 represents a node in the tree and each oval represents an attribute. To keep the figure simple, only the element and attribute names are included.

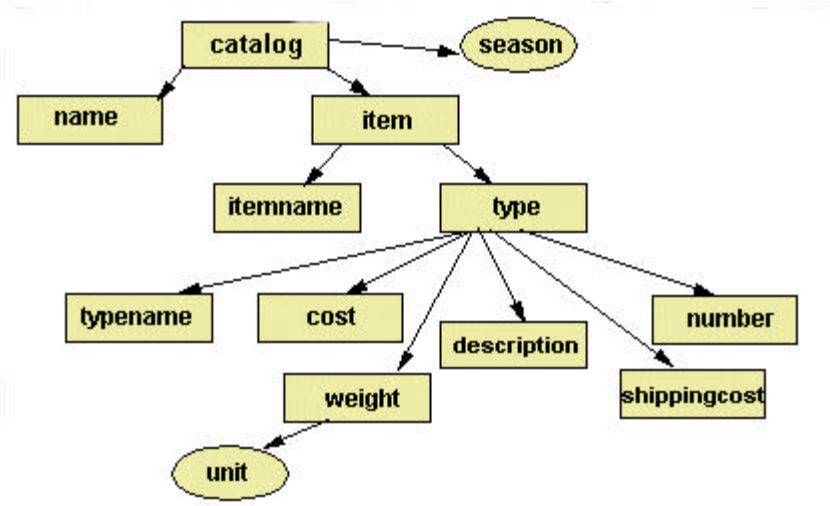

Figure 5 - Document Object Model

The DOM provides the methods to access the elements within the tree. By following the tree (hierarchical) structure, the methods allow traversing the tree using method calls for the parents and children. The DOM provides methods to traverse the tree created by the parser. The methods access the elements within the tree using the parent-child relationship.

The Example

This parsing example accomplishes the following activities:

1. create the DOM tree for the XML document
2. print the complete tree including attributes
3. find and replace the specified element value passed as parameters
4. print the complete (updated) tree.

This example uses the FindReplaceDOMParse class, the IBM XML4J parser and the DOM APIs to access and manipulate the XML Document. The import statements are shown in figure 6.

```
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Element;import org.w3c.dom.NamedNodeMap;
```

Figure 6 - import statements

The import statements listed in Figure 6 are part of the support for the Document Object Model API provided with the XML4J parser. More information can be found in the documentation provided with the XML4J parser.

Key portions of the main() method are shown in subsequent figures. The main() method is expecting four parameters:

1. srcfile — the XML document name
2. targetElement — element (name) involved in the find & replace
3. valueToFind — desired value to replace
4. valueToReplace — new value

Each of these parameters are converted to Strings within the main(). An example of the run time command to execute this program is:

java <directory path>.FindReplaceDOMParse catalog.xml cost $50.00 $95.00

The parameters specified in the command are in the order expected by the main() method.

The next step in the processing flow is the creation of the DOM tree for the catalog.xml document, which is accomplished by parsing the specified source file (srcFile). This parse is being done in validation mode, which ensures a valid document. Figure 7 contains the call statement to the method that parses the XML document.

```
    Document document = parseV(srcFile);
```

Figure 7 - call parseV method

The parseV() method, shown in Figure 8, does the following steps:

1. creates an instance of the parser
2. parse the XML document (srcfile)
3. get the DOM (Document object) and return it to the caller.

```
public static Document parseV(String srcFile) throws Exception{
        try {
                // Get a new parser and attach an error handler
                DOMParser myParser = new DOMParser();
                // Parse the file
                myParser.parse(srcFile);
                // Return the document
                return myParser.getDocument();
        } catch (Exception ex) {
                System.err.println("[Operation Terminated] " + ex);
        }
        return null;
}
```

Figure 8 - Parse the XML document to create a DOM tree

Figure 9 contains the method calls within the main() method for the processing of the document. The objective is to find the element (passed as the second parameter) and element value (passed as the third parameter) and change it to the element value (passed as the four parameter). Before the element is found and replaced, the DOM tree is printed and after the change, the tree is printed again.

```
if (document != null) {
        // Print the initial state of the document starting at the root element
        System.out.println("******************BEFORE******************");
        printElement(document.getDocumentElement());

           // Perform a find replace on the document
        document = findReplace(document, targetElement, valueToFind, valueToReplace);

           // Print the resulting state of the document starting at the root element
        System.out.println("******************AFTER******************");
          printElement(document.getDocumentElement());

   } else{
            System.out.println(" in main document null");
       }}
```

Figure 9 - main() method calls for printing

Here is a look at the methods called within the main() method that are doing the processing:

- **printElement(document.getDocumentElement())** which is being passed to the root of the document (DOM tree) by the getDocumentElement() method.
- **document = findReplace(document, targetElement, valueToFind, valueToReplace)** finds the element(s) to change and completes the change
- **printElement(document.getDocumentElement())** prints the updated XML document.

The first print version will look like the document shown in Figure 2 and using the command defined after Figure 6, the cost of the men's jacket will change from $50.00 to $95.00.

The printElement() method, shown in Figure 10, processes the complete document (DOM tree), which includes element names, element values, and attributes.

```
private static void printElement(Element element) {
 int k;
 NamedNodeMap attributes;
 NodeList children = element.getChildNodes();
 //***** Start this element
 System.out.print("<" + element.getNodeName());
 //***** Get any attibutes and print them inside the element start tag
 attributes = element.getAttributes();
 if (attributes != null) {
        for (k = 0; k < attributes.getLength(); k++) {
                System.out.print(" " + attributes.item(k).getNodeName());
                System.out.print("=" + attributes.item(k).getNodeValue());
        }
}
 if (element.hasChildNodes()) {  //***** If this element has a value or sub-elements
  System.out.print(">");
 //** For each child, if the child is an element call print element to print that portion
 //** of the tree, if it is a text node, print the text to stdout.
 //** All other node types are ignored for the sake of simplicity.

  for (k = 0; k < children.getLength(); k++) {
   if (children.item(k).getNodeType() == org.w3c.dom.Node.ELEMENT_NODE) {
        printElement((Element) children.item(k));
   } else if (children.item(k).getNodeType() == org.w3c.dom.Node.TEXT_NODE) {
        System.out.print(children.item(k).getNodeValue());
        }
  }     // end for loop
  //***** Add a closing tag
  System.out.print("");
}// end else
}// end method
```

Figure 10 - printElement() method

The printElement() method is recursive and continues to call itself until all the nodes in the tree are processed. The primary data type to deal with in the DOM interfaces is the Node interface.

The element passed to the printElement() is processed as follows:

1. getChildNodes() which returns a NodeList containing all the children for this node
2. use println to print the "<" and the element's Node name using getNodeName() method.
3. use getAttributes() method to return a NamedNodeMap containing the attributes of the node
4. process the attributes (if any exist) by looping through all attributes for the element
    a. print the attribute name via getNodeName()
    b. print an equal sign (=) and the attribute value via getNodeValue()
5. the NodeList children contains the children for this element, which was determined in step 1. If the element has children (determined by calling the hasChildNodes() method), each child is processed inside the loop. The node in the list is either an ELEMENT_NODE or a TEXT_NODE.
    a. An element node is processed by calling the printElement() method to ensure the element and its children are processed.
    b. A text node is simply printed because it is the node value rather than a new node.
6. Each node with its element information and attributes has an associated end tag.
7. For those elements which do not have an element value, the element name is printed as a empty element with the representation.

Take a look at step 5 for determining how to process the element. The code is checking for an ELEMENT_NODE or a TEXT_NODE in order to do the right type of processing. A particular node can have sub elements and/or text, as shown in figure 11, which is a representation of a phone number. The phone number depicts an element with two nodes; one being an element and the other is text for the element (phone number). Figure 11 helps visualize the possible processing in step 5.

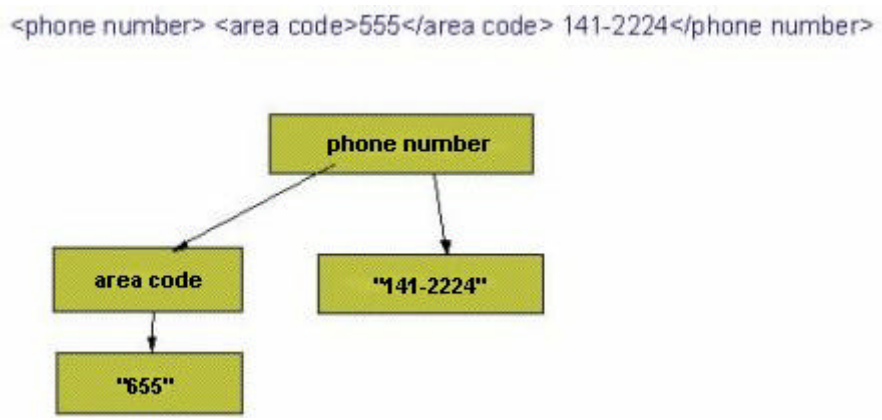

Figure 11 - Example DOM tree fragment

Once the original document is printed, the findReplace() method is called to modify the specified element's value. This method must perform similar functions as the printElement() method, except the parameters from the main() method are passed to this method to be processed. The document (DOM tree) is passed as a parameter. Figure 12 contains the code for processing the find and replace activity. The new methods are in bold to show the new capabilities being explored.

Because much of this code is similar to the code within the printElement() method, only the highlighted code for the findReplace() method is discussed. The highlighted code is discussed in ascending order:

- from the root element, the element method getElementsByTagName() method is used to retrieve the elements (the results are placed in a NodeList).
- once the node in the NodeList is determined to be a TEXT_NODE, the node is compared against the passed in value to determine if this is an element to be changed. This is accomplished by using the equals() method on this node within the NodeList.
- use the setNodeValue() method to set the value with the new value

```
static private Document findReplace(Document document, String elementName, String
valueToFind, String valueToReplace) {
  int i;     int k;
  NodeList children;
 Element docRoot = document.getDocumentElement();  //***  Get the root element

  NodeList elements = docRoot.getElementsByTagName(elementName);
  if (elements !=  null) {
     //** For each element matching the search element
    for (i = 0; i<elements.getLength(); i++) {
        if (elements.item(i).hasChildNodes()) {
            children = elements.item(i).getChildNodes();
           for (k = 0; k<children.getLength(); k++) {
                if (children.item(k).getNodeType() == org.w3c.dom.Node.TEXT_NODE) {
                  if (children.item(k).getNodeValue().equals(valueToFind)) {
                        children.item(k).setNodeValue(valueToReplace);
                  } //* end if
                } //* end if
            } //* end for loop
        } //* end if
      }//* end for loop
  }// end if
  return document;
} // end method
```

Figure 12 - findReplace() method

The only method within the class left to discuss is the usage() method which is called when an error in processing occurs. This method consists of a println that prints the values passed on the call to the class.

Summary

This tutorial provided an example of using the XML parser (XML4J) and a variety of the capabilities provided by the parser and the DOM APIs to access and update a XML document. More information on XML4J, the classes and methods associated with it are included in the documentation when you download the parser.

Trademarks

About the Author:

LindaMay Patterson
PartnerWorld for Developers, AS/400
Advisory Software Engineer

LindaMay Patterson is an Advisory Programming in PartnerWorld for Developers, AS/400 at IBM Rochester. She has worked at IBM for 24 years in various Business Application environments. Currently, she is part of the PartnerWorld for Developers, AS/400 Java team working with Enterprise JavaBeans. Prior to this, she worked on the SanFrancisco product developing the education packages and working on SanFrancisco in Germany helping to define the product content. Her Application background is primarily in Distribution and Logistics Systems.

For more information on PartnerWorld for Developers, AS/400:

To get more details on becoming a member of PartnerWorld for Developers and AS/400, visit our Web site at:
www.ibm.com/as400/developer/membership/reg_info.html

Check out our PartnerWorld for Developers, AS/400 service offerings at:
www.ibm.com/as400/developer/java/

Production

Teresa Powell PartnerWorld for Developers, AS/400 Member Services