Understanding DOM

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of Contents

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

Section 1. Introduction

Should I take this tutorial?

This tutorial is designed for developers who understand the basic concept of XML and are ready to move on to coding applications to manipulate XML using the Document Object Model (DOM). It assumes that you are familiar with concepts such as well-formedness and the tag-like nature of an XML document. (If necessary, you can get a basic grounding in XML itself through the *Introduction to XML* tutorial.)

All of the examples in this tutorial are in Java, but you can develop a thorough understanding of the DOM through this tutorial even if you don't try out the examples yourself. The concepts and API for coding an application that manipulates XML data in the DOM are the same for any language or platform, and no GUI programming is involved.

Tools

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

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

JavaTM 2 SDK, Standard Edition version 1.3.1: The sample applications demonstrate manipulation of the DOM through Java. You can download the Java SDK from *http://java.sun.com/j2se/1.3/* .

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

Other Languages: Should you wish to adapt the examples, DOM implementations are also available in other programming languages. You can download C++ and Perl implementations of a DOM parser called Xerces from the Apache Project at *http://xml.apache.org* .

Conventions used in this tutorial

* Text that needs to be typed is displayed in a **`bold monospace`** font. Bold is used in some code examples to draw attention to a tag or element being referenced in the accompanying text.
* *Emphasis/Italics* is used to draw attention to windows, dialog boxes and feature names.
* A `monospace` font presents file and path names.
* Throughout this tutorial, code segments irrelevant to the discussion have been omitted and replaced with ellipses (`...`).

About the author

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

Section 2. What is the Document Object Model?

The foundation of XML

The Document Object Model (DOM) is the foundation of Extensible Markup Language, or XML. XML documents have a hierarchy of informational units called *nodes*; DOM is a way of describing those nodes and the relationships between them.

In addition to its role as a conceptual description of XML data, the DOM is a series of Recommendations maintained by the World Wide Web Consortium (W3C). It began as a way to allow Web browsers to identify and manipulate elements on a page -- functionality that predates W3C involvement and is referred to as "DOM Level 0".

The actual DOM Recommendation, which is currently at Level 2 (with Level 3 expected to become a Recommendation toward the beginning of 2002), is an API that defines the objects that are present in an XML document and the methods and properties that are used to access and manipulate them.

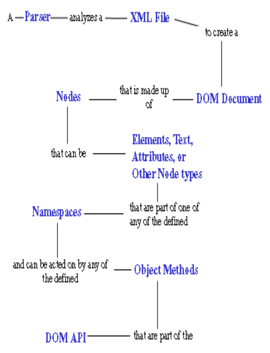
This tutorial demonstrates the use of the DOM Core API as a means for reading and manipulating XML data using the example of a series of orders from a commerce system. It also teaches you how to create DOM objects in your own projects for the purposes of storing or working with data.

The DOM as structure

Before beginning work with the DOM, it pays to have an idea of what it actually represents. A DOM `Document` is a collection of *nodes*, or pieces of information, organized in a hierarchy. This hierarchy allows a developer to navigate around the tree looking for specific information. Analyzing the structure normally requires that you load the entire document and load the hierarchy before any work is done. Because it is based on a hierarchy of information, the DOM is said to be *tree based*.

For exceptionally large documents, parsing and loading the entire document can be slow and resource intensive, so there are other means for dealing with the data. These *event-based* models, such as the Simple API for XML (SAX), work on a stream of data, processing it as it goes by. (SAX is the subject of another tutorial and other articles in the developerWorks XML zone. See Resources on page 31for more information.) An event-based API eliminates the need to build the data tree in memory, but it doesn't allow a developer to actually change the data in the original document.

The DOM, on the other hand, also provides an API that allows a developer to add, edit, move, or remove nodes at any point on the tree in order to create an application.



A DOM roadmap

Working with the DOM involves several concepts that all fit together. We'll examine these relationships in the course of this tutorial.

A *parser* is a software application that is designed to analyze a document -- in this case an XML file -- and do something specific with the information. In an event-based API like SAX, the parser sends events to a listener of some sort. In a tree-based API like DOM, the parser builds the data tree in memory.

The DOM as an API

Starting with DOM Level 1, the DOM API contains interfaces that represent all of the different types of information that can be found in an XML document, such as elements and text. It also includes the methods and properties necessary to work with these objects.

Level 1 included support for XML 1.0 and HTML, with each HTML element represented as an interface. It included methods for adding, editing, moving, and reading the information contained in nodes, and so on. It did not, however, include support for XML Namespaces, which provide the ability to segment information within a document.

Namespace support was added to the DOM Level 2. Level 2 extends Level 1, allowing developers to detect and use the namespace information that might be applicable for a node. Level 2 also adds several new modules supporting Cascading Style Sheets, events, and enhanced tree manipulations.

DOM Level 3 is still being written, but is expected to add enhanced namespace support, including: two new recommendations, XML Infoset and XML Base; extended support for user interface events and support for DTD; XML Schema loading and saving abilities; and other features. Notably, it also adds support for XPath, the means used in XSL Transformations to locate specific nodes.

The modularization of the DOM means that as a developer, you must know whether the features you wish to use are supported by the DOM implementation you are working with.

Determining feature availability

The modular nature of the DOM Recommendations allows implementors to pick and choose which sections to include in their product, so it may be necessary to determine whether a particular feature is available before attempting to use it. This tutorial will use only the DOM Level 2 Core API, but it pays to understand how features can be detected when moving on in your own projects.

`DOMImplementation` is one of the interfaces defined in the DOM. By using the `hasFeature()` method, you can determine whether a particular feature is supported. No standard way of creating a `DOMImplementation` exists, but the following code demonstrates how to use `hasFeature()` to determine whether the DOM Level 2 Style Sheets module is supported in a Java application using JAXP 1.1. The concepts are the same for other implementations.

```
import javax.xml.parsers.DocumentBuilderFactory;
 import javax.xml.parsers.DocumentBuilder;
 import org.w3c.dom.DOMImplementation;

 public class ShowDomImpl {

    public static void main (String args[]) {
       try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          DocumentBuilder docb = dbf.newDocumentBuilder();
          DOMImplementation domImpl = docb.getDOMImplementation();

          if (domImpl.hasFeature("StyleSheets", "2.0")) {
             System.out.println("Style Sheets are supported.");
          } else {
             System.out.println("Style Sheets are not supported.");
          }
       } catch (Exception e) {}
         }

 }
```

The tutorial will use a single document to demonstrate the objects and methods of the DOM Level 2 Core API.

The basic XML file

Examples throughout this tutorial use an XML file that contains the code example below, which represents orders working their way through a commerce system. To review, the basic parts of an XML file are:

* The XML declaration: The basic declaration `<?xml version"1.0"?>` defines this file as an XML document. It's not uncommon to specify an encoding in the declaration, as shown below. This way, it doesn't matter what language or encoding the XML file uses, the parser will be able to read it properly as long as it understands that particular encoding.
* The DOCTYPE declaration: XML is a convenient way of exchanging information between humans and machines, but a common vocabulary is necessary for XML to work smoothly. The optional DOCTYPE declaration can be used to specify a document (in this case, `orders.dtd`), against which this file should be compared to ensure there is no stray or missing information (for example, a missing `userid` or misspelled element name). Documents processed this way are known as *valid* documents. Successful validation is not a requirement for XML.
* The data itself: Data in an XML document must be contained within a single *root element*, such as the `orders` element below. In order for an XML document to be processed, it must be *well formed*.

```
                     <?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE ORDERS SYSTEM "orders.dtd">
 <orders>
    <order>
       <customerid limit="1000">12341</customerid>
       <status>pending</status>
       <item instock="Y" itemid="SA15">
          <name>Silver Show Saddle, 16 inch</name>
```

```
         <price>825.00</price>
         <qty>1</qty>
      </item>
      <item instock="N" itemid="C49">
         <name>Premium Cinch</name>
         <price>49.00</price>
         <qty>1</qty>
      </item>
   </order>
   <order>
      <customerid limit="150">251222</customerid>
      <status>pending</status>
      <item instock="Y" itemid="WB78">
         <name>Winter Blanket (78 inch)</name>
         <price>20</price>
         <qty>10</qty>
      </item>
   </order>
</orders>
```

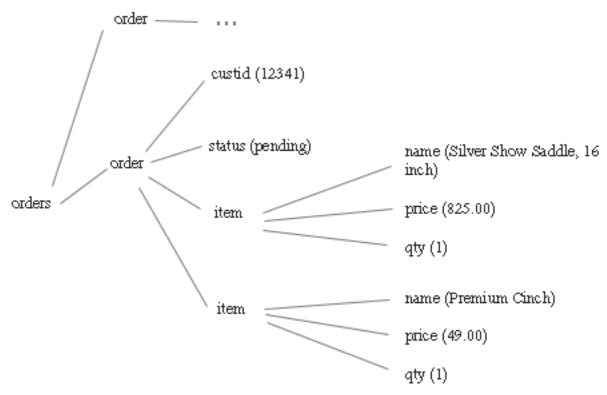
In the DOM, working with XML information means first breaking it down into nodes.

Section 3. The different types of XML nodes

Creating the hierarchy

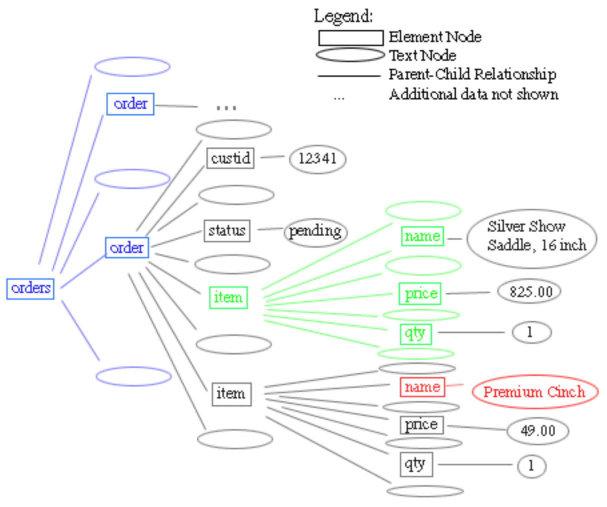
DOM, in essence, is a collection of nodes. With different types of information potentially contained in a document, there are several different types of nodes defined.

In creating a hierarchy for an XML file, it's natural to produce something conceptually like the structure below. While it is an accurate depiction of the included data, it is not an accurate description of the data as represented by the DOM. This is because it represents the *elements*, but not the *nodes*.



The difference between elements and nodes

In fact, elements are only one type of node, and they don't even represent what the previous diagram seems to indicate. An *element node* is a container for information. That information may be other element nodes, text nodes, attribute nodes, or other types of information. A more accurate picture of the document is below.



The rectangular boxes represent element nodes, and the ovals represent *text nodes*. When one node is contained within another node, it is considered to be a *child* of that node.

Notice that the `orders` element node has not two, but five children: two `order` elements, and the text nodes between and around them. Even though there is no content, the whitespace between `order` elements makes up a text node. Similarly, `item` has seven children: `name`, `price`, `qty`, and the four text nodes around them.

Notice also that what might be considered the content of an element, such as "Premium Cinch", is actually the content of a text node that is the child of the `name` element.

(Even this diagram is not complete, leaving out, among other things, the attribute nodes.)

The basic node types: Document, Element, Attribute, and Text

The most common types of nodes in XML are:

* Elements: Elements are the basic building blocks of XML. Typically, elements have children that are other elements, text nodes, or a combination of both. Element nodes are also the only type of node that can have attributes.
* Attributes: Attribute nodes contain information about an element node, but are not actually considered to be children of the element, as in:

```
<customerid>12341</customerid>
```

* Text: A text node is exactly that: text. It can consist of more information or just white space.
* Document: The document node is the overall parent for all of the other nodes in the document.

Less common node types: CData, Comment, Processing Instructions, and Document Fragments

Other node types are less frequently used, but still essential in some situations. They include:

* CData: Short for Character Data, this is a node that contains information that should not be analyzed by the parser. Instead, it should just be passed on as plain text. For example, HTML might be stored for a specific purpose. Under normal circumstances, the processor might try to create elements for each of the stored tags, which might not even be well-formed. These problems can be avoided by using CData sections. CData sections are written with a special notation:

```
<[CDATA[<b>Important:  Please keep head and hands inside ride at
<i>all times</i>.</b>]]>
```

* Comment: Comments include information about the data, and are usually ignored by the application. They are written as:

```
<!--  This is a comment. -->
```

* Processing Instructions: PIs are information specifically aimed at the application. Some examples are code to be executed or information on where to find a style sheet. For example:

```
<?xml-stylesheet type="text/xsl" href="foo.xsl"?>
```

* Document Fragments: In order to be well formed, a document can have only one root element. Sometimes, groups of elements must be temporarily created that don't necessarily fulfill this requirement. A document fragment looks like this:

```
<item instock="Y" itemid="SA15">
    <name>Silver Show Saddle, 16 inch</name>
    <price>825.00</price>
    <qty>1</qty>
 </item>
 <item instock="N" itemid="C49">
    <name>Premium Cinch</name>
    <price>49.00</price>
    <qty>1</qty>
 </item>
```

Other types of nodes include entities, entity reference nodes, and notations.

One way of further organizing XML data is through the use of Namespaces.

Section 4. Namespaces

What is a namespace?

One of the main enhancements between the DOM Level 1 and the DOM Level 2 is the addition of support for *Namespaces*. Namespace support allows developers to use information from different sources or with different purposes without conflicts.

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

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

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

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

Creating a namespace

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

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE orders SYSTEM "orders.dtd">
 <orders xmlns="http://www.nicholaschase.com/orderSystem.html">
      <order>
       <customerid limit="1000">12341<customerid>
 ...
 </orders>
```

(Again, the `...` indicates sections that aren't relevant.)

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

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

Designating namespaces

Other namespaces can also be designated for data. For example, by creating a `rating` namespace you can add credit rating information to the text of the orders without disturbing the actual data.

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

Consider the code below. The namespace and the alias, `rating`, have been used to create the `creditRating` element. Because the `creditRating` element is in a separate namespace, the fact that it isn't listed in the DTD is irrelevant.

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE orders SYSTEM "orders.dtd">
 <orders xmlns="http://www.nicholaschase.com/orderSystem.html"
       xmlns:rating="http://www.nicholaschase.com/rating.html">
    <order>
       <customerid limit="1000">
          12341
          <rating:creditRating>good</rating:creditRating>
       </customerid>
       <status>
          pending
       </status>
       <item instock="Y" itemid="SA15">
          <name>
             Silver Show Saddle, 16 inch
          </name>
          <price>
             825.00
          </price>
          <qty>
             1
          </qty>
       </item>
    ...
 </orders>
```

Namespace information can be obtained for a node after the document has been parsed.

Section 5. Parsing a file into a document

The three-step process

In order to work with the information in an XML file, the file must be parsed to create a `Document` object.

The `Document` object is an interface, so it can't be instantiated directly; generally, the application uses a factory instead. The exact process varies from implementation to implementation, but the ideas are the same. In the example JAXP environment parsing the file is a three-step process:

1. Create the `DocumentBuilderFactory`. This object will create the `DocumentBuilder`.
2. Create the `DocumentBuilder`. The `DocumentBuilder` will do the actual parsing to create the `Document` object.
3. Parse the file to create the `Document` object.

JAXP allows you to plug in a different parser, if desired, without having to change your code. Let's go ahead and start building the application.

The basic application

Start by creating the basic application, a class called `OrderProcessor`.

```
import javax.xml.parsers.DocumentBuilder;
 import javax.xml.parsers.DocumentBuilderFactory;
 import java.io.File;
 import org.w3c.dom.Document;

 public class OrderProcessor {
    public static void main (String args[]) {
       File docFile = new File("orders.xml");
       Document doc = null;
       try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          DocumentBuilder db = dbf.newDocumentBuilder();
          doc = db.parse(docFile);
       } catch (Exception e) {
          System.out.print("Problem parsing the file.");
       }
    }
 }
```

First, Java imports the necessary classes, then creates the `OrderProcessor` application. The examples in this tutorial will deal with one file, so for brevity's sake, the application contains a direct reference to it.

So that the `Document` object can be used later, the application defines it outside the `try-catch` block. A `try-catch` block lets you perform actions that could potentially throw an exception without risking the entire application. If an exception is thrown, the application simply executes the appropriate `catch` code.

Within the `try-catch` block, the application creates the
, then uses it to create the `DocumentBuilder`. Finally, the `DocumentBuilder` parses the file to create the `Document`.

Parser settings

One of the advantages of creating parsers with a `DocumentBuilder` is the control over various settings on the parsers created by the `DocumentBuilderFactory`. For example, the parser can be set to validate the document:

```
...
 try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          dbf.setValidating(true);
          DocumentBuilder db = dbf.newDocumentBuilder();
          doc = db.parse(docFile);
       } catch (Exception e) {
 ...
```

JAXP allows control over the following parameters for a parser:

* setCoalescing: Determines whether the parser will turn CDATA nodes into text, and merge them with surrounding text nodes (if applicable). The default value is `false`.
* setExpandEntityReferences: Determines whether external entity references will be expanded. If `true`, the external data will be inserted into the document. The default value is `true`. (See Resources on page 31for tips on working with external entities.)
* setIgnoringcomments: Determines whether comments within the file will be ignored. The

default value is `false`.
* setIgnoringElementContentWhitespace: Determines whether whitespace within element contents will be ignored (similar to the way a browser treats HTML). The default value is `false`.
* setNameSpaceAware: Determines whether the parser will pay attention to namespace information. The default value is `false`.
* setValidating: By default, parsers won't validate documents. Set this to `true` to turn on validation.

Parser exceptions

With all of the different possibilities in creating a parser, there are many different things that can go wrong. As the example stands, the application dumps all of them into a single, generic `Exception`, which might not be very helpful in terms of debugging.

In order to better pinpoint problems, you can catch specific exceptions related to the various aspects of creating and using a parser. For example, in the code below, notice that if the application throws an `IOException`, it will output "Cannot read input file." If, on the other hand, it throws a `SAXException`, the application will output "Problem parsing the file."

```
...
       try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          DocumentBuilder db = dbf.newDocumentBuilder();
          doc = db.parse(docFile);
       } catch (javax.xml.parsers.ParserConfigurationException pce) {
          System.out.println("The parser was not configured correctly.");
          System.exit(1);
       } catch (java.io.IOException ie) {
          System.out.println("Cannot read input file.");
          System.exit(1);
       } catch (org.xml.sax.SAXException se) {
          System.out.println("Problem parsing the file.");
          System.exit(1);
       } catch (java.lang.IllegalArgumentException ae) {
          System.out.println("Please specify an XML source.");
          System.exit(1);
       }
 ...
```

Once the parser has created a `Document`, the application can step through it to examine the data.

Section 6. Stepping through the document

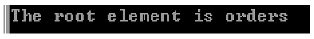
Get the root element

Once the document is parsed and a `Document` is created, an application can step through the structure to review, find, or display information,. This navigation is the basis for many operations that will be performed on a `Document`.

Stepping through the document begins with the root element. A well-formed document has only one root element, also known as the `DocumentElement`. First the application retrieves this element.

Compiling and running the application outputs the name of the root element, `orders`.

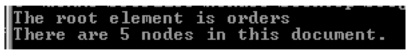
```
import javax.xml.parsers.DocumentBuilder;
 import javax.xml.parsers.DocumentBuilderFactory;
 import java.io.File;
 import org.w3c.dom.Document;
 import org.w3c.dom.Element;

 public class OrderProcessor {
 ...
          System.exit(1);
       }

       //STEP 1:  Get the root element
       Element root = doc.getDocumentElement();
       System.out.println("The root element is " + root.getNodeName());
    }
 }
```



```
The root element is orders
```

Get the children of a node

Once the application determines the root element, it retrieves a list of the root element's children as a `NodeList`. The `NodeList` class is a series of items that the application can iterate through. In this example, for brevity, the application gets the child nodes and verifies the retrieval by showing only how many elements appear in the resulting `NodeList`.



```
The root element is orders
There are 5 nodes in this document.
```

Notice that the document has only two elements, but the `NodeList` contains five children, including three text nodes that contain line feeds -- another reminder that nodes and elements are not equivalent in the DOM. The other three nodes are text nodes containing line feeds.

```
...
 import org.w3c.dom.NodeList;
 ...
    //STEP 1:  Get the root element
    Element root = doc.getDocumentElement();
    System.out.println("The root element is "+root.getNodeName());

    //STEP 2:  Get the children
    NodeList children = root.getChildNodes();
    System.out.println("There are "+children.getLength()
                                 +" nodes in this document.");
  }
 }
```

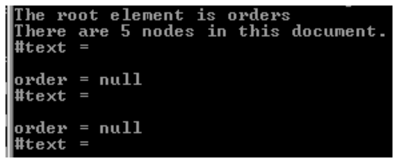
Using getFirstChild() and getNextSibling()

The parent-child and sibling relationships offer an alternative means for iterating through all of the children of a node that may be more appropriate in some situations, such as when these relationships and the order in which children appear is crucial to understanding the data.

In Step 3, a `for-loop` starts with the first child of the root. The application iterates through each of the siblings of the first child until all of them have been evaluated.

Each time the application executes the loop, it retrieves a `Node` object, outputting its name and value. Notice that the five children of orders include the `order` elements and three text nodes. Notice also that the elements carry a value of null, rather than the expected text: The text nodes that are children of the elements carry the actual content as their values.

```
...
 import org.w3c.dom.Node;
 ...

       //STEP 3:  Step through the children
       for (Node child = root.getFirstChild();
           child != null;
           child = child.getNextSibling())
       {
          System.out.println(child.getNodeName()+" = "+child.getNodeValue());
       }
    }
 }
 ...
```



```
The root element is orders
There are 5 nodes in this document.
#text =

order = null
#text =

order = null
#text =
```

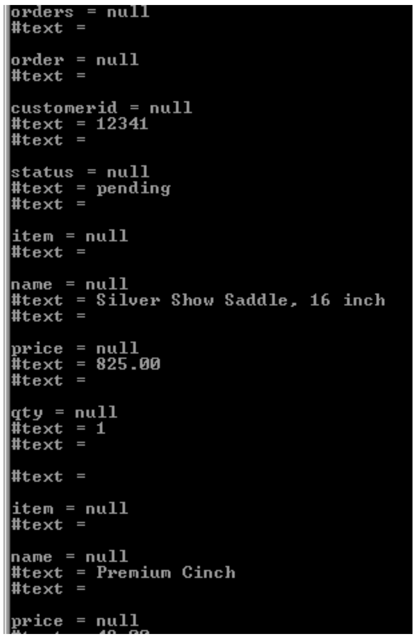
Recursing through multiple levels of children

The code in the previous panel shows the first-level children, but that's hardly the entire document. In order to see all of the elements, the functionality in the previous example must be turned into a method and called recursively.

The application starts with the root element and prints the name and value to the screen. The application then runs through each of its children, just as before. But for each of the children, the application also runs through each of their children, examining all of the children and grandchildren of the root element.

```
...
 public class OrderProcessor {

    private static void stepThrough (Node start)
```

```
   {
      System.out.println(start.getNodeName()+" = "+start.getNodeValue());

      for (Node child = start.getFirstChild();
          child != null;
          child = child.getNextSibling())
      {
            stepThrough(child);
      }
   }

   public static void main (String args[]) {
      File docFile = new File("orders.xml");

...
      System.out.println("There are "+children.getLength()
                            +" nodes in this document.");

      //STEP 4:  Recurse this functionality
      stepThrough(root);
   }
}
```



```
orders = null
#text =

order = null
#text =

customerid = null
#text = 12341
#text =

status = null
#text = pending
#text =

item = null
#text =

name = null
#text = Silver Show Saddle, 16 inch
#text =

price = null
#text = 825.00
#text =

qty = null
#text = 1
#text =

#text =

item = null
#text =

name = null
#text = Premium Cinch
#text =

price = null
```

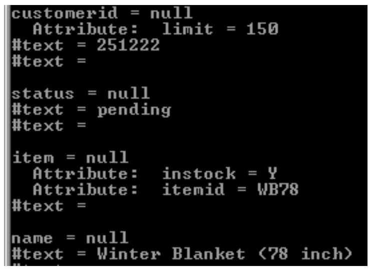
Including attributes

The `stepThrough()` method as written so far can run through most types of nodes, but it misses attributes entirely because they are not children of any nodes. To show attributes, modify `stepThrough()` to check element nodes for attributes.

The modified code checks each node output to see whether or not it's an element by comparing its `nodeType` to the constant value `ELEMENT_NODE`. A `Node` object carries member constants that represent each type of node, such as `ELEMENT_NODE` or `ATTRIBUTE_NODE`. If the `nodeType` matches `ELEMENT_NODE`, it is an element.

For every element it finds, the application creates a `NamedNodeMap`, which contains all of the attributes for the element. The application can iterate through a `NamedNodeMap`, printing each one's name and value, just as it iterated through the `NodeList`.

```
...
 import org.w3c.dom.NamedNodeMap;
 ...
 private static void stepThroughAll (Node start)
    {
       System.out.println(start.getNodeName()+" = "+start.getNodeValue());

       if (start.getNodeType() == start.ELEMENT_NODE)
       {
           NamedNodeMap startAttr = start.getAttributes();
           for (int i = 0;
                i < startAttr.getLength();
                i++) {
              Node attr = startAttr.item(i);
              System.out.println("  Attribute:  "+ attr.getNodeName()
                                             +" = "+attr.getNodeValue());
           }
       }

       for (Node child = start.getFirstChild();
           child != null;
           child = child.getNextSibling())
       {
          stepThroughAll(child);
       }
    }
```



```
customerid = null
  Attribute:  limit = 150
#text = 251222
#text =

status = null
#text = pending
#text =

item = null
  Attribute:  instock = Y
  Attribute:  itemid = WB78
#text =

name = null
#text = Winter Blanket (78 inch)
```

Section 7. Editing the document

Changing the value of a node

Reviewing the contents of an XML `Document` is useful, but when dealing with full-featured applications you may need to change the data (add, edit, move or delete information). The ability to edit data is also crucial to the process of creating new XML `Documents`. The simplest such change is one that effects the text content of an element.

The goal here is to change the value of an element's text node, in this case by setting the `status` of each `order` to "processed," then printing the new values to the screen.

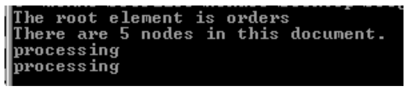
The `changeOrder()` method is called with the starting node (`root`) as a parameter, as well as both the name of the element to change and the value to change it to.

`changeOrder()` first checks the name of the node to see if it is one of the elements to edit. If it is, the application needs to change the value not of this node, but of its first child, as this first child will be the text node that actually contains the content.

In either case, the application checks each child, just as it did in stepping through the document the first time.

When the changes are complete, the values are checked by using `getElementsByTagName()`. This method returns a list of all child elements with a specific name, such as `status`. The application can then check the list for values to verify that `changeOrder()` worked.

```
...
 public class OrderProcessor {
    private static void changeOrder (Node start,
                                String elemName,
                               String elemValue)
    {
       if (start.getNodeName().equals(elemName)) {
          start.getFirstChild().setNodeValue(elemValue);
       }

       for (Node child = start.getFirstChild();
           child != null;
           child = child.getNextSibling())
       {
           changeOrder(child, elemName, elemValue);
       }
    }
 ...
    public static void main (String args[]) {
 ...

      // Change text content
       changeOrder(root, "status", "processing");
       NodeList orders = root.getElementsByTagName("status");
       for (int orderNum = 0;
            orderNum < orders.getLength();
            orderNum++)
       {
           System.out.println(orders.item(groupNum).getFirstChild().getNodeValue());
       }
    }
 }
```



```
The root element is orders
There are 5 nodes in this document.
processing
processing
```

Notice that the application picked up the `status` nodes even though they are grandchildren of the root element, and not direct children. `getElementsByTagName()` will step through the document and find all elements with a particular name.

Adding nodes: prepare the data

Rather than changing an existing node, it is sometimes necessary to add a node, and there are several ways to do it. In this example, the application totals the cost for each `order` and adds a `total` element.

It obtains the total by taking each order and looping through each of its items to get the item cost, then totaling them. The application then adds the new element to the order.

First, the application retrieves the `order` elements just as it retrieved the status elements. It then loops though each of these elements.

For each of these `orders`, the application needs a `NodeList` of its `item` elements, so the application must first cast the `order` `Node` to `Element` in order to use `getElementsByTagName()`.

The application can then loop through the `item` elements for the selected `order`. For each one, it casts to `Element` so that it can retrieve the `price` and `qty` by name. It does that by using `getElementsByTagName()`, and, because there is only one of each, it can go directly to `item(0)`, which is the first entry in the resulting `NodeList`. This first entry represents the `price` (or `qty`) element. From there the value of the text node is obtained.

The value of the text node is a `String` value, which the application then converts to a `double` to allow the necessary math.

When the application has finished examining all of the items for each order, `total` is a `double` that represents the total value. The `total` is then converted to a `String` so it can be used as the content of a new element, `total`, which ultimately joins the `order`.

```
...
 changeOrder(root, "status", "processing");
 NodeList orders = root.getElementsByTagName("order");
 for (int orderNum = 0;
      orderNum < orders.getLength();
      orderNum++)
 {
    Element thisOrder = (Element)orders.item(orderNum);
    NodeList orderItems = thisOrder.getElementsByTagName("item");
    double total = 0;
    for (int itemNum = 0;
         itemNum < orderItems.getLength();
         itemNum++) {

       // Total up cost for each item and
       // add to the order total

         //Get this item as an Element
         Element thisOrderItem = (Element)orderItems.item(itemNum);

         //Get pricing information for this Item
         String thisPrice = thisOrderItem.getElementsByTagName("price").item(0)
                                .getFirstChild().getNodeValue();
         double thisPriceDbl = new Double(thisPrice).doubleValue();
```

```
        //Get quantity information for this Item
        String thisQty = thisOrderItem.getElementsByTagName("qty").item(0)
                               .getFirstChild().getNodeValue();
        double thisQtyDbl = new Double(thisQty).doubleValue();

        double thisItemTotal = thisPriceDbl*thisQtyDbl;
        total = total + thisItemTotal;
    }
    String totalString = new Double(total).toString();
 }
 ...
```

Adding nodes: add nodes to the document

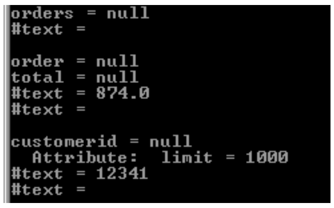
There are multiple ways to create a new `Node`, and this example uses several of them. First, the `Document` object can create a new text node, with `totalString` as the value. The new `Node` now exists, but isn't actually attached to the `Document` anywhere. The new `total` element is similarly created, and is that is also disembodied to start with.

Another way to add a node is to use `appendChild()`, as seen here in adding the node to the new `total` element.

Finally, the application can use `insertBefore()` to add the new element to the `Document`, specifying the new `Node`, then the `Node` that it precedes.

Stepping through the document verifies the changes.

```
...
 changeOrder(root, "status", "processing");
 NodeList orders = root.getElementsByTagName("order");
 for (int orderNum = 0;
      orderNum < orders.getLength();
      orderNum++)
 {
 ...
    String totalString = new Double(total).toString();
    Node totalNode = doc.createTextNode(totalString);

    Element totalElement = doc.createElement("total");
    totalElement.appendChild(totalNode);

    thisOrder.insertBefore(totalElement, thisOrder.getFirstChild());
 }

 stepThrough(root);

 ...
```



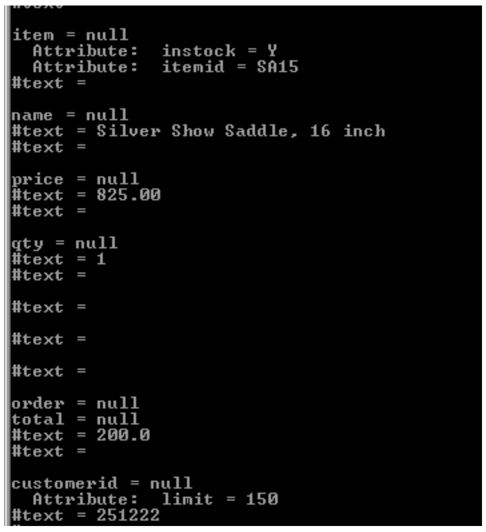
```
orders = null
#text =

order = null
total = null
#text = 874.0
#text =

customerid = null
  Attribute:  limit = 1000
#text = 12341
#text =
```

Remove a node

Rather than replacing the text of an element, the application can remove it altogether. In this example, the application will check to see if the item is in stock. If not, it deletes the item from the order instead of adding it to the total.

Before adding an item's cost it to the total, the application checks the value of the `instock` attribute. If it is `N`, then instead of being added it to the total, the item is removed completely. To do that, it uses the `removeChild()` method, but first determines `orderItem`'s parent node using `getParentNode()`. The actual `Node` is removed from the document, but the method also returns it as an object, so it could be moved if desired.

```
...
    //Get this item as an Element
    Element thisOrderItem = (Element)orderItems.item(itemNum);

    if (thisOrderItem.getAttributeNode("instock").getNodeValue().equals("N")) {

       Node deadNode = thisOrderItem.getParentNode().removeChild(thisOrderItem);

    } else {

       //Get pricing information for this Item
       String thisPrice = thisOrderItem.getElementsByTagName("price").item(0)
                                        .getFirstChild().getNodeValue();
 ...
       total = total + thisItemTotal;
    }
 }
 String totalString = new Double(total).toString();

 ...
```



```
item = null
  Attribute:  instock = Y
  Attribute:  itemid = SA15
#text =

name = null
#text = Silver Show Saddle, 16 inch
#text =

price = null
#text = 825.00
#text =

qty = null
#text = 1
#text =

#text =

#text =

#text =

order = null
total = null
#text = 200.0
#text =

customerid = null
  Attribute:  limit = 150
#text = 251222
```

Replace a node

Of course, it doesn't make much sense to remove an item because it's backordered. Instead, the application will replace it with a `backordered` item.

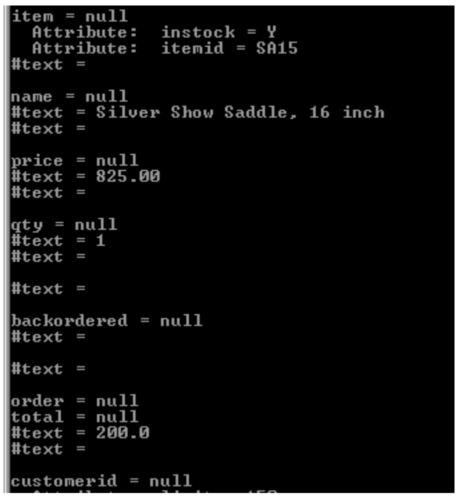
Instead of using `removeChild()`, simply use `replaceChild()`. Notice that in this case, the method also returns the old node, so it can be moved, if necessary, perhaps to a new `Document` listing backordered items.

Notice that because no content was added to the element, it is an empty element. An empty element has no content, and is written with a special shorthand:

```
<backordered />
```

The slash (/) eliminates the need for an end tag (`</backordered>`).

```
...
    if (thisOrderItem.getAttributeNode("instock").getNodeValue().equals("N")) {

       Element backElement = doc.createElement("backordered");
       Node deadNode = thisOrderItem.getParentNode()
                          .replaceChild(backElement,
                                            thisOrderItem);


    } else {
 ...
```



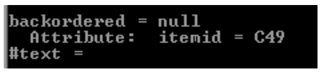
```
item = null
  Attribute:  instock = Y
  Attribute:  itemid = SA15
#text =

name = null
#text = Silver Show Saddle, 16 inch
#text =

price = null
#text = 825.00
#text =

qty = null
#text = 1
#text =

#text =

backordered = null
#text =

#text =

order = null
total = null
#text = 200.0
#text =

customerid = null
```

Creating and setting attributes

Of course, what good does a `backordered` element do if there's no indication of what item it is? One way to correct the lack of information is by adding an attribute to the element.

First the application creates an `itemid` attribute. Next, it determines the value of `itemid` from the original `item` element, then sets the value of the attribute itself. Finally, it adds the element to the document, just as before.

```
...
 if (thisOrderItem.getAttributeNode("instock").getNodeValue().equals("N")) {

    Element backElement = doc.createElement("backordered");

    backElement.setAttributeNode(doc.createAttribute("itemid"));

    String itemIdString = thisOrderItem.getAttributeNode("itemid").getNodeValue();
    backElement.setAttribute("itemid", itemIdString);

    Node deadNode = thisOrderItem.getParentNode().replaceChild(backElement,
                                                             thisOrderItem);

 } else {
 ...
```



```
backordered = null
  Attribute:  itemid = C49
#text =
```

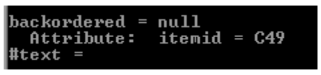
It's important to note that `setAttribute()` will create the attribute node if it doesn't exist, so in this case, the application could have skipped `createAttribute()` entirely.

Removing an attribute

An application can also remove attributes. For example, it might not be advisable for customer credit limit information to show in the output, so the application could temporarily remove it from the document.

Removing the information is fairly straightforward, using `removeAttribute()` to remove the data.

```
...
 Element thisOrder = (Element)orders.item(orderNum);

 Element customer = (Element)thisOrder.getElementsByTagName("cusomertid").item(0);
 customer.removeAttribute("limit");

 NodeList orderItems = thisOrder.getElementsByTagName("item");
 ...
```



```
backordered = null
  Attribute:  itemid = C49
#text =
```

The next step utilizes the limit information, however, so remove the latest change before moving on.

Section 8. Outputting the document

Preparing the data

The tutorial so far has looked at taking in, working with, and manipulating XML data. To complete the cycle, you must also be able to output XML.

In the case of this tutorial, the target output is a file that simply lists each order, whether it was processed or rejected based on the customer's credit limit, and the `customerid`.

```
<?xml version="1.0" encoding="UTF-8"?>
 <processedOrders>
    <order>
           <status>PROCESSED</status>
           <customerid>2341</customerid>
           <amount>874.00</amount>
    </order>
    <order>
           <status>REJECTED</status>
           <customerid>251222</customerid>
           <amount>200.00</amount>
    </order>
 </processedOrders>
```

First the application creates the `Document` object to output. For convenience, the same `DocumentBuilder` that created the original `Document` can create the new one.

After processing `orders.xml`, the application creates a new element, `processedOrders`, which will eventually become the root element of the new document. It then runs through each of the orders. For each one, it extracts the `total` and `limit` information.

Next the application creates the new elements for the order: `order`, `status`, `customerid`, and `amount`. It populates `status` based on whether the total exceeds the customer's credit limit, and populates the others accordingly.

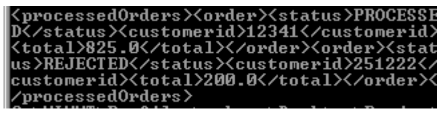
Once the application creates the elements, it must put them together. First it adds the status, customer information, and total to the new `order` element. Then it adds the new `order` to the `newRoot` element.

While all this is going on, the `newRoot` element is not actually attached to a parent node. When the application has completed processing all orders, `newRoot` is appended to the new document.

Finally, the application outputs the data by converting `newRoot` to a `String` and simply sending it to `System.out`.

```
...
    public static void main (String args[]) {
       File docFile = new File("orders.xml");
       Document doc = null;
           Document newdoc = null;
       try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          DocumentBuilder db = dbf.newDocumentBuilder();
          doc = db.parse(docFile);

                  newdoc = db.newDocument();

       } catch (...) {
 ...
```

```
Element newRoot = newdoc.createElement("processedOrders");

NodeList processOrders = doc.getElementsByTagName("order");
for (int orderNum = 0;
     orderNum < processOrders.getLength();
     orderNum++) {

   Element thisOrder = (Element)processOrders.item(orderNum);

   Element customerid = (Element)thisOrder.getElementsByTagName("customerid")
                                          .item(0);
   String limit = customerid.getAttributeNode("limit").getNodeValue();

   String total = thisOrder.getElementsByTagName("total").item(0)
                        .getFirstChild().getNodeValue();

   double limitDbl = new Double(limit).doubleValue();
   double totalDbl = new Double(total).doubleValue();

   Element newOrder = newdoc.createElement("order");

   Element newStatus = newdoc.createElement("status");
   if (totalDbl > limitDbl) {
      newStatus.appendChild(newdoc.createTextNode("REJECTED"));
   } else {
      newStatus.appendChild(newdoc.createTextNode("PROCESSED"));
   }

   Element newCustomer = newdoc.createElement("customerid");
   String oldCustomer = customerid.getFirstChild().getNodeValue();
   newCustomer.appendChild(newdoc.createTextNode(oldCustomer));

   Element newTotal = newdoc.createElement("total");
   newTotal.appendChild(newdoc.createTextNode(total));

   newOrder.appendChild(newStatus);
   newOrder.appendChild(newCustomer);
   newOrder.appendChild(newTotal);

   newRoot.appendChild(newOrder);
}

newdoc.appendChild(newRoot);

System.out.print(newRoot.toString());
...
```



```
<processedOrders><order><status>PROCESSE
D</status><customerid>12341</customerid>
<total>825.0</total></order><order><stat
us>REJECTED</status><customerid>251222</
customerid><total>200.0</total></order><
/processedOrders>
```

Normalized vs. non-normalized data

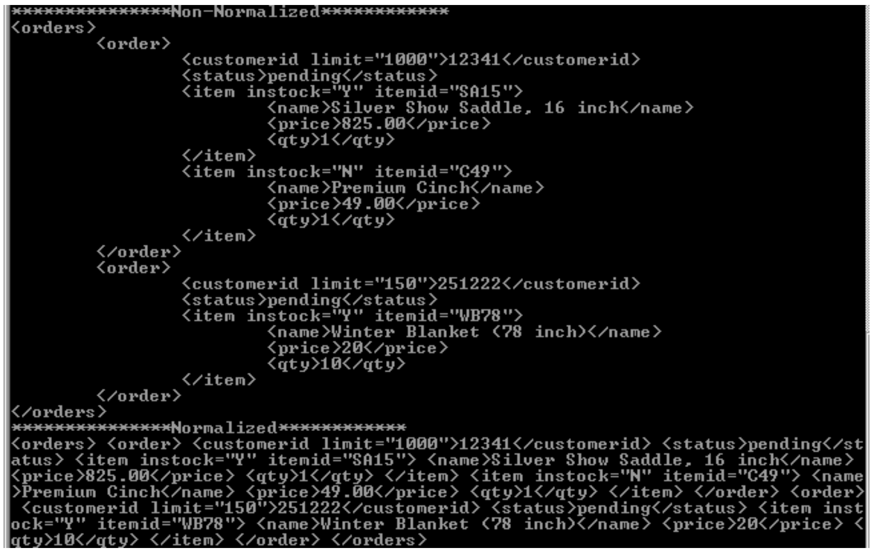
You may notice that rather than the nice hierarchical structure that is often seen in an XML file, the data shown in the previous panel is all on a single (wrapped) line. XML data without extraneous text nodes is called *normalized data*, and is convenient for ensuring an application will see the data the same way every time. Remember, all of those line feeds and indents enter into the document as nodes, and can dramatically alter the data's apparent structure.

Fortunately, an application can normalize any node using the `normalize()` method. Consider the following:

```
System.out.println("***************Non-Normalized************");
 System.out.println(root.toString());
```

```
        System.out.println("***************Normalized************");
        root.normalize();
        System.out.println(root.toString());
```

This example code will output the following:



```
***************Non-Normalized************
<orders>
        <order>
                <customerid limit="1000">12341</customerid>
                <status>pending</status>
                <item instock="Y" itemid="SA15">
                        <name>Silver Show Saddle, 16 inch</name>
                        <price>825.00</price>
                        <qty>1</qty>
                </item>
                <item instock="N" itemid="C49">
                        <name>Premium Cinch</name>
                        <price>49.00</price>
                        <qty>1</qty>
                </item>
        </order>
        <order>
                <customerid limit="150">251222</customerid>
                <status>pending</status>
                <item instock="Y" itemid="WB78">
                        <name>Winter Blanket (78 inch)</name>
                        <price>20</price>
                        <qty>10</qty>
                </item>
        </order>
</orders>
***************Normalized************
<orders> <order> <customerid limit="1000">12341</customerid> <status>pending</st
atus> <item instock="Y" itemid="SA15"> <name>Silver Show Saddle, 16 inch</name>
<price>825.00</price> <qty>1</qty> </item> <item instock="N" itemid="C49"> <name
>Premium Cinch</name> <price>49.00</price> <qty>1</qty> </item> </order> <order>
 <customerid limit="150">251222</customerid> <status>pending</status> <item inst
ock="Y" itemid="WB78"> <name>Winter Blanket (78 inch)</name> <price>20</price> <
qty>10</qty> </item> </order> </orders>
```

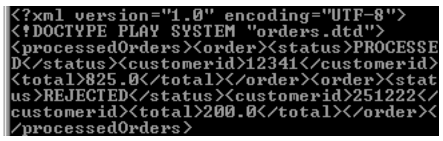
When a node is normalized, adjacent text nodes are merged into one, and empty text nodes (such as those between elements) are removed.

The DOCTYPE declaration

In reality, an XML file is more than just the data it contains. In outputting an XML document you must be certain to add the XML declaration and the DOCTYPE. Fortunately, information can be retrieved from the original DOCTYPE in order to reproduce it for the new document. For now, the application sends output to `System.out`.

First it outputs the XML declaration, then the DOCTYPE declaration. The application adds the original DOCTYPE to the document itself as a node, so it can retrieve the information from doc using `getDocType()`. From here, it can retrieve the name and the public or system ID, as appropriate.

```
...
 System.out.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
 System.out.print("<!DOCTYPE "+doc.getDoctype().getName()+" ");
 if (doc.getDoctype().getSystemId() != null)
 {
    System.out.print(" SYSTEM ");
    System.out.print(doc.getDoctype().getSystemId());
 }
 if (doc.getDoctype().getPublicId() != null)
 {
    System.out.print(" PUBLIC ");
    System.out.print(doc.getDoctype().getPublicId());
 }
 System.out.println(">");

 System.out.println(newRoot.toString());
 ...
```



```
<?xml version="1.0" encoding="UTF-8">
<!DOCTYPE PLAY SYSTEM "orders.dtd">
<processedOrders><order><status>PROCESSE
D</status><customerid>12341</customerid>
<total>825.0</total></order><order><stat
us>REJECTED</status><customerid>251222</
customerid><total>200.0</total></order><
/processedOrders>
```

Creating an XML File

Now that the application has created the new information, outputting it to a file is straightforward.

The same logic is used for the data, but instead of outputting it to the screen, the application sends it to a file.

The important thing to note here is that since XML data is just text, it can be formatted in any way. For example, you could create a variation on `stepThroughAll()` that would create an indented, or "pretty printed" version. Just remember that the extra whitespace (text) nodes that will be created.

```
...
 import java.io.FileWriter;
 ...
 try
 {
    File newFile = new File("processedOrders.xml");
    FileWriter newFileStream = new FileWriter(newFile);

    newFileStream.write("<?xml version=\"1.0\"?>");
    newFileStream.write("<!DOCTYPE "+doc.getDoctype().getName()+" ");
    if (doc.getDoctype().getSystemId() != null)
    {
       newFileStream.write(" SYSTEM ");
       newFileStream.write(doc.getDoctype().getSystemId());
    }
    if (doc.getDoctype().getPublicId() != null)
    {
       newFileStream.write(" PUBLIC ");
       newFileStream.write(doc.getDoctype().getPublicId());
    }
    newFileStream.write(">");

    newFileStream.write(newRoot.toString());

    newFileStream.close();

 } catch (IOException e) {
     System.out.println("Can't write new file.");
 }
 ...
```

Other output options

Outputting to a file is only one way to use XML data. Some others include:

Streaming: An application can stream XML data to a SAX parser or other application. SAX, or the Simple API for XML, is another way of handling XML data. A SAX parser reads each character of a stream and allows an application to use it to build a model of the document.

SOAP: SOAP, or the Simple Object Access Protocol, is a method for transmitting information and instructions from one machine to another using HTTP. The information is encoded in specifically formatted XML.

Transformations: XML can be transformed using Extensible Stylesheet Language Transformations, or XSLT. An XSL processor returns a transformed version of the data, ready for other purposes. For example, the application could have used XSL to create the final report of approved and rejected orders.

Section 9. Document Object Model summary

Summary

The DOM is a language- and platform-independent API designed for working with XML data. It is a tree-based API that loads all of the data into memory as a parent-child hierarchy of nodes, which may be elements, text, attributes, or other node types.

The DOM API allows a developer to read, create, and edit XML data. This tutorial talked about the concepts behind the DOM and illustrated them with examples in Java. Implementations of DOM are also available in C++, Perl, and other languages.

Resources

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

* For a basic grounding in XML read through the *Introduction to XML* tutorial.
* Read *G. Ken Holman's article* for an in-depth discussion of XSLT.
* Read *Norman Walsh's article* at XML.COM for more information about working with external entities.
* The *W3C DOM page* is a very useful DOM resource, containing topics such as DOM Level 2 erratum, W3C DOM activity statements, test suites, and related sources.
* Order , by Nicholas Chase, the author of this tutorial.
* *The XML Cover Pages* contains specifications on DOM Levels 1, 2 and 3, as well as links to articles and other DOM references.
* Download *the Java 2 SDK* , Standard Edition version 1.3.1.
* Download *JAXP 1.1* , the JavaTM APIs for XML Processing.
* Download *Xerces C++* , a validating DOM parser.
* Download *Xerces.pm* , a Perl API implemented using the Xerces C++ API, which provides access to most of the C++ API from Perl.

Feedback

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

Colophon

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