Understanding DOM

Skill Level: Introductory

Nicholas Chase (nicholas@nicholaschase.com)
Author, Web site developer

29 Jul 2003

Even before there was XML, there was the Document Object Model, or DOM. It allows a developer to refer to, retrieve, and change items within an XML structure, and is essential to working with XML. In this tutorial, you will learn about the structure of a DOM document. You will also learn how to use Java technology to create a Document from an XML file, make changes to it, and retrieve the output.

Section 1. Tutorial 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. (You can get a basic grounding in XML itself through the Introduction to XML tutorial, if necessary.)

All of the examples in this tutorial are in the Java language, 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.

What is the Document Object Model?

The foundation of Extensible Markup Language, or XML, is the DOM. 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). The DOM began as a way for Web browsers to identify and manipulate elements on a page -- functionality that predates the W3C's involvement and is referred to as "DOM Level 0".

The actual DOM Recommendation, which is currently at Level 2 (with Level 3 in Last Call status as of this writing), is an API that defines the objects that are present in an XML document, as well as 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 storing or working with data.

Tools

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

- **A text editor:** XML files are simply text. To create and read them, a text editor is all you need.
- **Java 2 SDK, Standard Edition version 1.4.x:** DOM support has been built into the latest version of Java technology (available at http://java.sun.com/j2se/1.4.2/download.html), so you won't need to install any separate classes. (If you're using an earlier version of the Java language, such as Java 1.3.x, you'll also need an XML parser such as the Apache project's Xerces-Java (available at http://xml.apache.org/xerces2-j/index.html), or Sun's Java API for XML Parsing (JAXP), part of the Java Web Services Developer Pack (available at http://java.sun.com/webservices/downloads/webservicespack.html).
- **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 the Xerces parser from the Apache Project at http://xml.apache.org.

Section 2. What is the DOM?

The DOM as structure

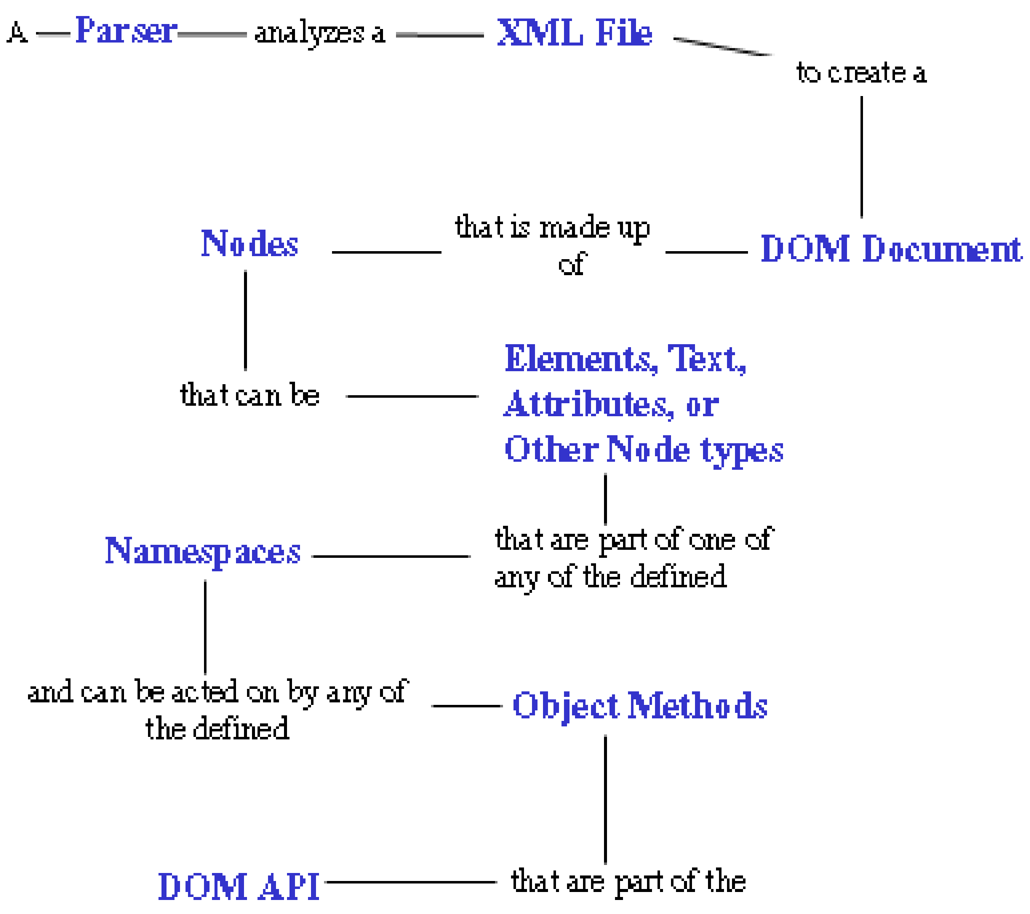
Before beginning work with the DOM, it pays to have an idea of what it actually represents. A DOM `Document` is a collection of **nodes**, or pieces of information, organized in a hierarchy. This hierarchy allows a developer to navigate around the tree looking for specific information. Analyzing the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

For exceptionally large documents, parsing and loading the entire document can be slow and resource-intensive, so other means are better 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 for more information.) An event-based API eliminates the need to build the data tree in memory, but 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 road map

Working with the DOM involves several concepts that all fit together. You'll learn about 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 a data tree in memory.

The DOM as an API

Starting with the 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, currently in Last Call, includes better support for creating a `Document` object (previous versions left this up to the implementation, which made it difficult to create generic applications), enhanced Namespace support, and new modules to deal with loading and saving documents, validation, and XPath, the means for selecting nodes used in XSL Transformations and other XML technologies.

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 with which you are working.

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 uses only the DOM Level 2 Core API, but it pays to understand how features can be detected when moving on in your own projects.

One of the interfaces defined in the DOM is `DOMImplementation`. By using the `hasFeature()` method, you can determine whether or not a particular feature is supported. No standard way of creating a `DOMImplementation` exists in DOM Level 2, 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.

```
 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) {}
        }

}
```

(DOM Level 3 will include a standard means for creating a `DOMImplementation`.)

This tutorial uses 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 is 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 for it to work smoothly, a common vocabulary is necessary. 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 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, and I'll actually leave the DOCTYPE declaration out of the document in subsequent examples.
- **The data itself:** Data in an XML document must be contained within a single **root element**, such as the `orders` element below. 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>
```

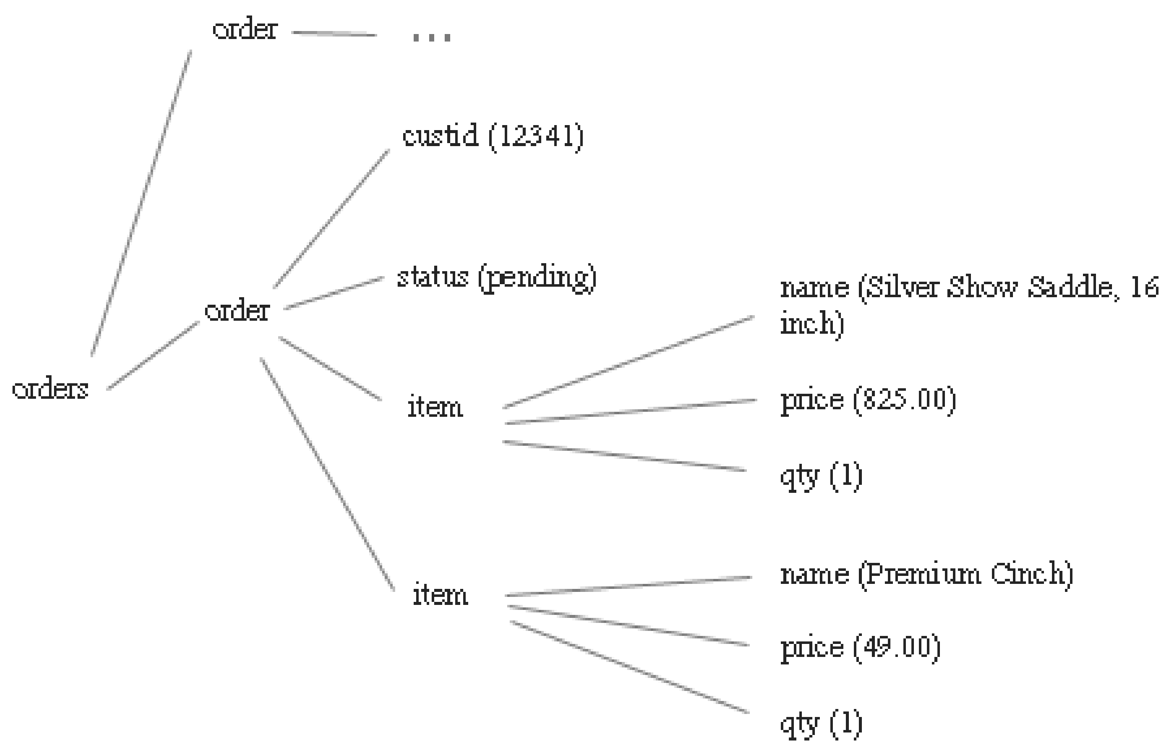
In the DOM, working with XML information means first breaking it down into nodes.

Section 3. The different types of XML nodes

Creating the hierarchy

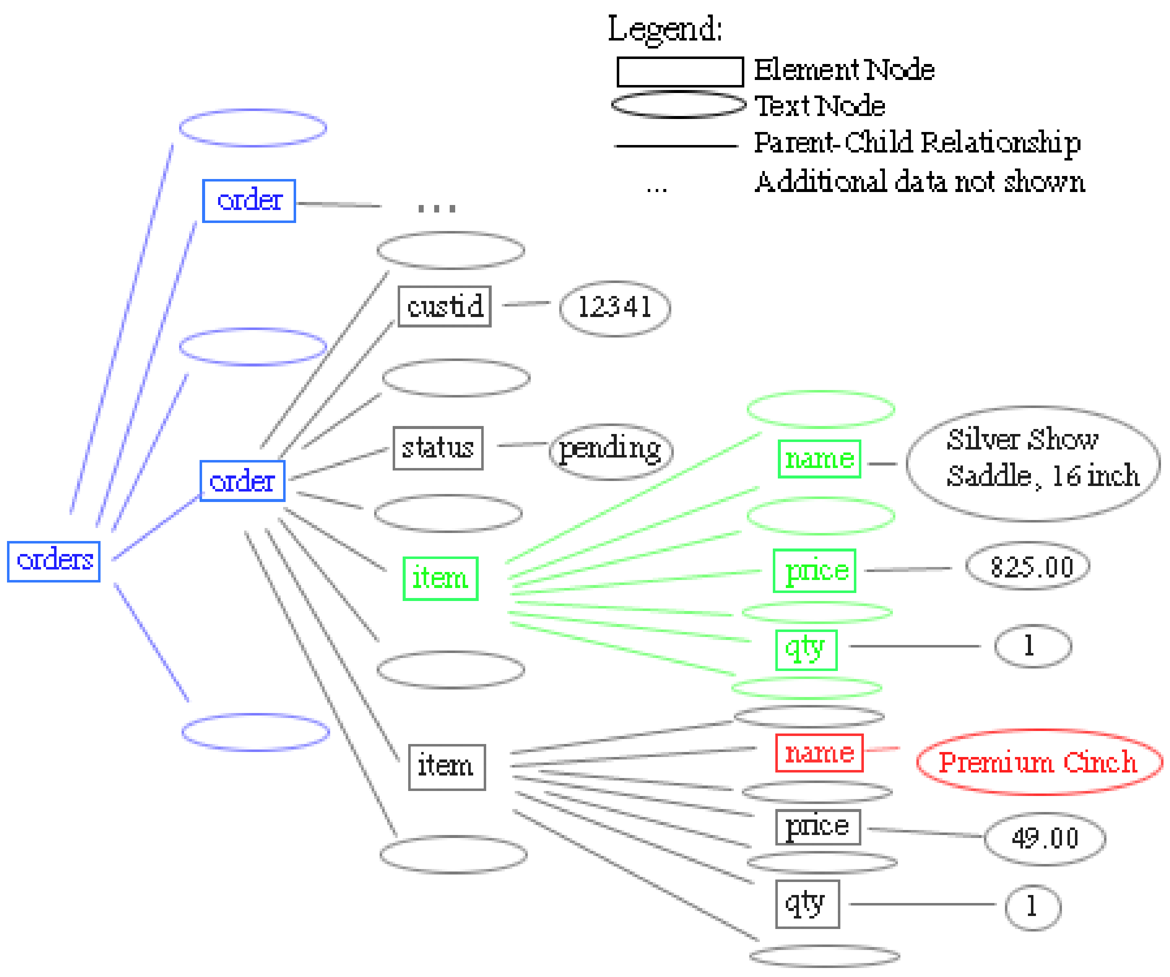
DOM, in essence, is a collection of nodes. With different types of information potentially contained in a document, several different types of nodes are defined.

In creating a hierarchy for an XML file, it is 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 it has no content, the white space 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 to be 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  limit="1000" >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. These 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 stylesheet. For example:

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

- **Document Fragments:** To be well-formed, a document can have only one root element. Sometimes, in working with XML, groups of elements must be temporarily created that don't necessarily fulfill this requirement. A document fragment might look 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 the caller was outside the office namespace.

On the other hand, if I were out of the office and she made the same announcement, confusion is likely 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"?>
<orders  xmlns="http://www.nicholaschase.com/orderSystem.html" >
     <order>
      <customerid limit="1000">12341<customerid>
...
</orders>
```

(The `...` indicates sections that aren't relevant.)

Any elements 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. Information may or may not be at that address, but what is important is that it is unique.

It's important to note the huge distinction between the default namespace and no namespace at all. In this case, elements that don't have a namespace prefix are in the default namespace. Previously, when no default namespace existed, those elements were in no namespace.

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

Designating namespaces

Other namespaces can also be designated for data. For example, 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.

```
 <?xml version="1.0" encoding="UTF-8"?>
<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

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. (Again, Level 3 standardizes this task.) In the example Java environment, parsing the file is a three-step process:

1. **Create the `DocumentBuilderFactory`.** This object creates the `DocumentBuilder`.
2. **Create the `DocumentBuilder`.** The `DocumentBuilder` does the actual parsing to create the `Document` object.
3. **Parse the file** to create the `Document` object.

Now you can 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: "+e.getMessage());
      }
   }
}
```

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

So the `Document` object can be used later, the application defines it outside the `try-catch` block.

Within the `try-catch` block, the application creates the `DocumentBuilderFactory`, which it then uses 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` lies in 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) {
...
```

Java's DOM Level 2 implementation allows control over the parameters for a parser through the following methods:

- **`setCoalescing()`:** Determines whether the parser turns CDATA nodes into text, and merges them with surrounding text nodes (if applicable). The default value is `false`.
- **`setExpandEntityReferences()`:** Determines whether external entity references are expanded. If `true`, the external data is inserted into the document. The default value is `true`. (See Resources for tips on working with external entities.)
- **`setIgnoringComments()`:** Determines whether comments within the file are ignored. The default value is `false`.
- **`setIgnoringElementContentWhitespace()`:** Determines whether whitespace within element contents is ignored (similar to the way a browser treats HTML). The default value is `false`.
- **`setNamespaceAware()`:** Determines whether the parser pays 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, many different things 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.

To better pinpoint problems, you can catch specific exceptions related to the various aspects of creating and using a parser:

```
...

    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

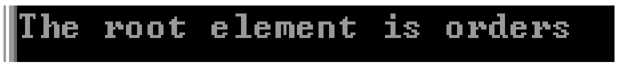
Get the root element

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.

```
 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());

   }
}
```

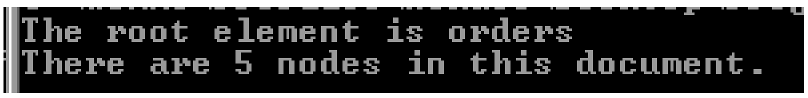
Compiling and running the application outputs the name of the root element, `orders`.



```
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 through which the application can iterate. 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.");

 }
}
```

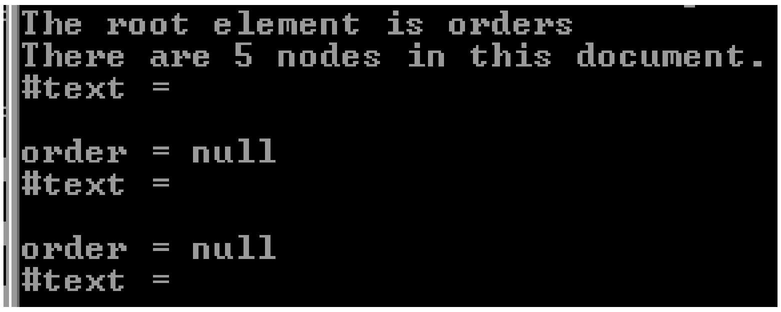
Using getFirstChild() and getNextSibling()

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 they have all 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. It is the text nodes that are children of the elements that 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(start.getNodeName()+" = "
                              +start.getNodeValue());
        }

      }
    }
    ...
```



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

order = null
#text =

order = null
#text =
```

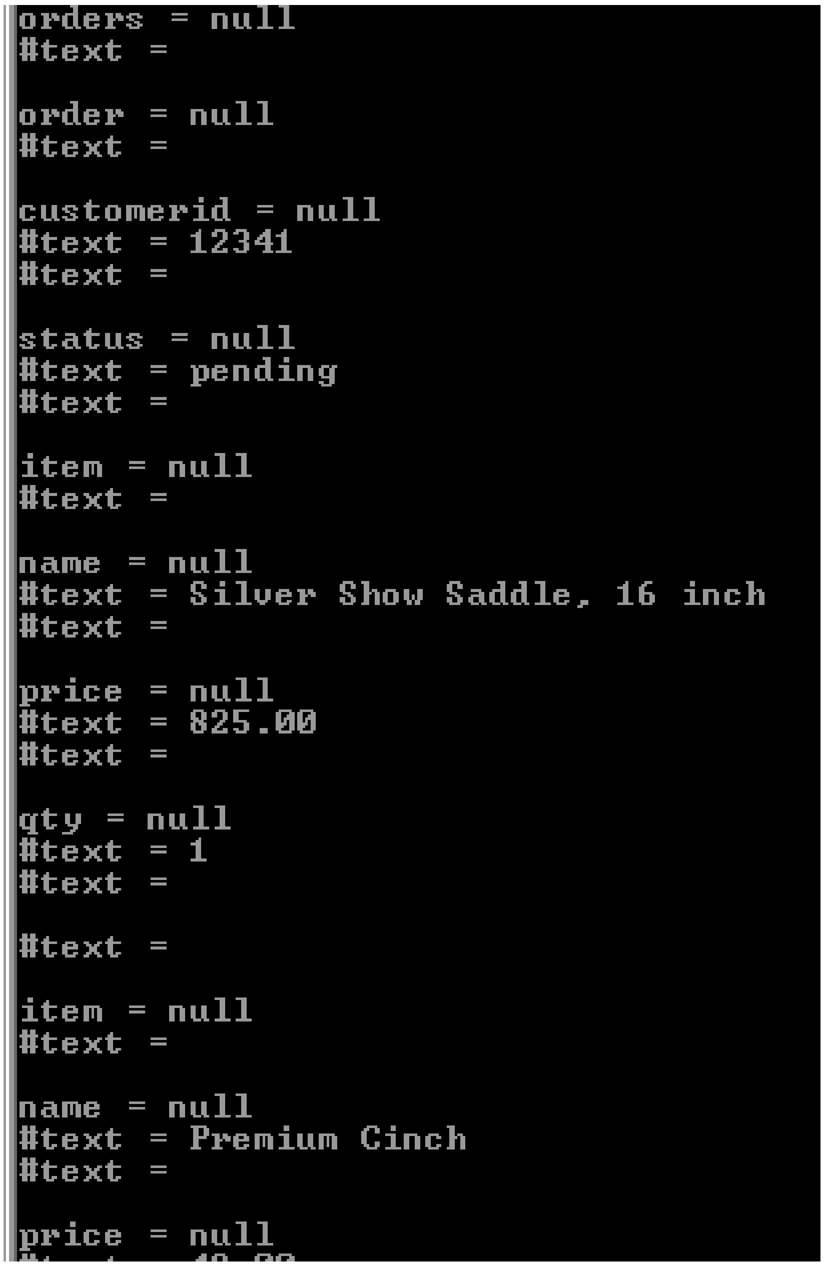
Recursing through multiple levels of children

The code in Using getFirstChild() and getNextSibling() on page shows the first-level children, but that's hardly the entire document. 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
```

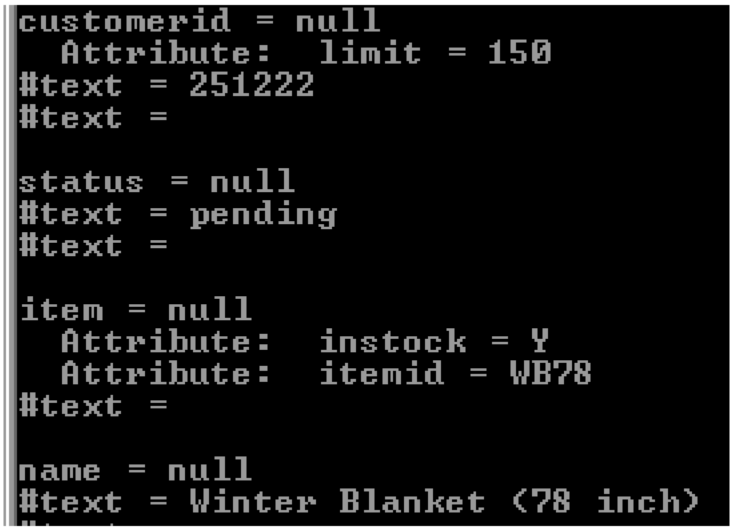
Including attributes

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 below 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 that contains all of the attributes for the element. The application can iterate through a NamedNodeMap, printing each attribute'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 to 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 affects 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", and 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.

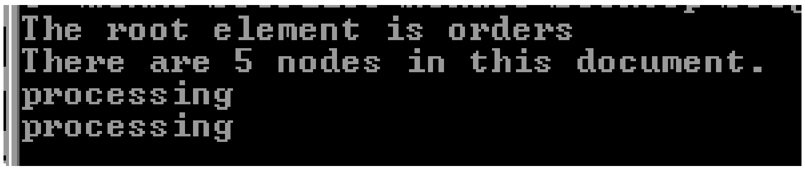
`changeOrder()` first checks the name of the node to see if it is one of the elements to edit. If it is, the application needs to change the value of not this node, but its first child, as this first child is the text node that actually contains the content.

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(orderNum)
                 .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()` steps 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 you can do this several ways. 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, and then totals them. The application then adds the new element to the order (see code listing below).

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 `order` s, the application needs a `NodeList` of its `item` elements, so the application must first cast the `order` `Node` to `Element` 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)`, 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

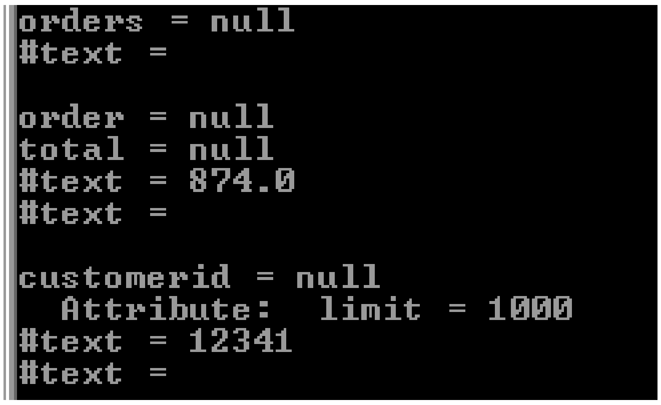
You can create a new `Node` in a number of ways, and this example uses several of them. First, the `Document` object can create a new text node, with `totalString` as the value. The new `Node` now exists, but isn't actually attached to the `Document` anywhere. The new `total` element is similarly created, and 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`, and 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);

...
```



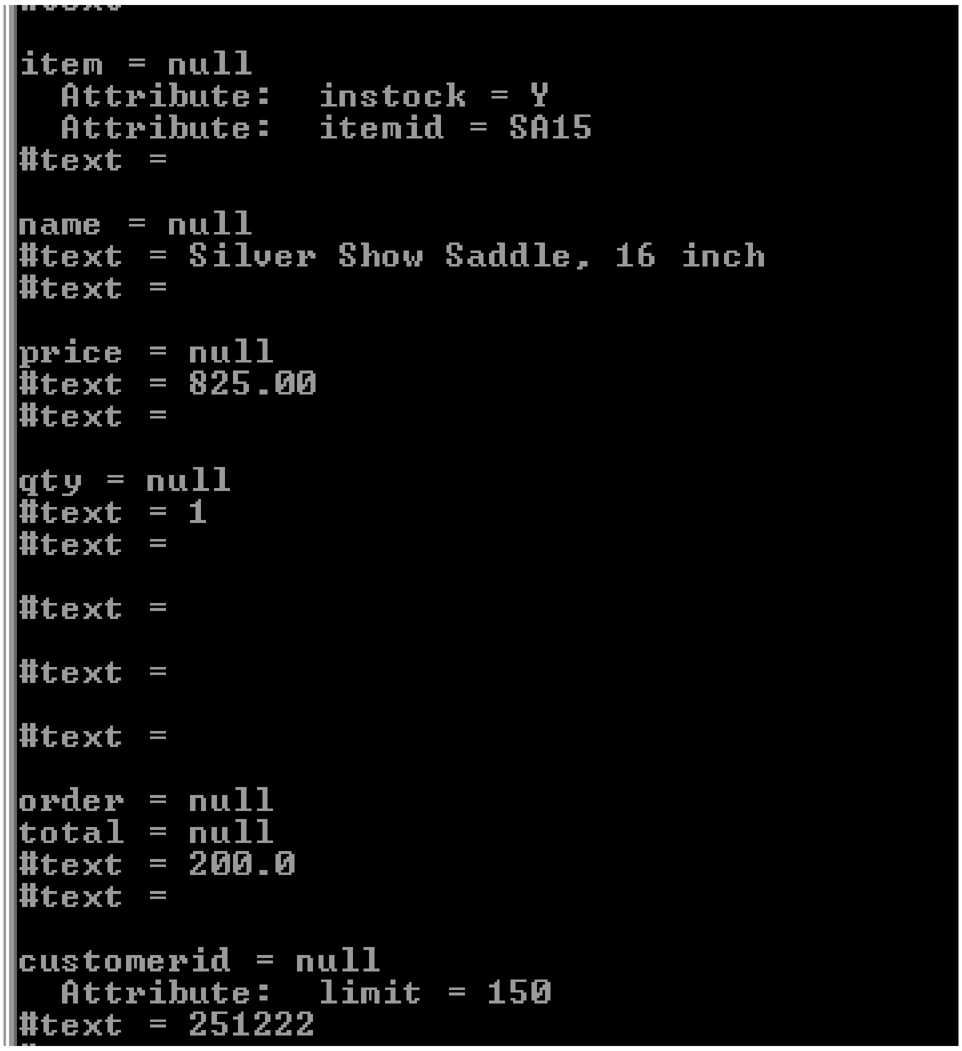
```
orders = null
#text =

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 checks 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 to the total, the application checks the value of the `instock` attribute. If it is `N`, then instead of being added 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 replaces 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.

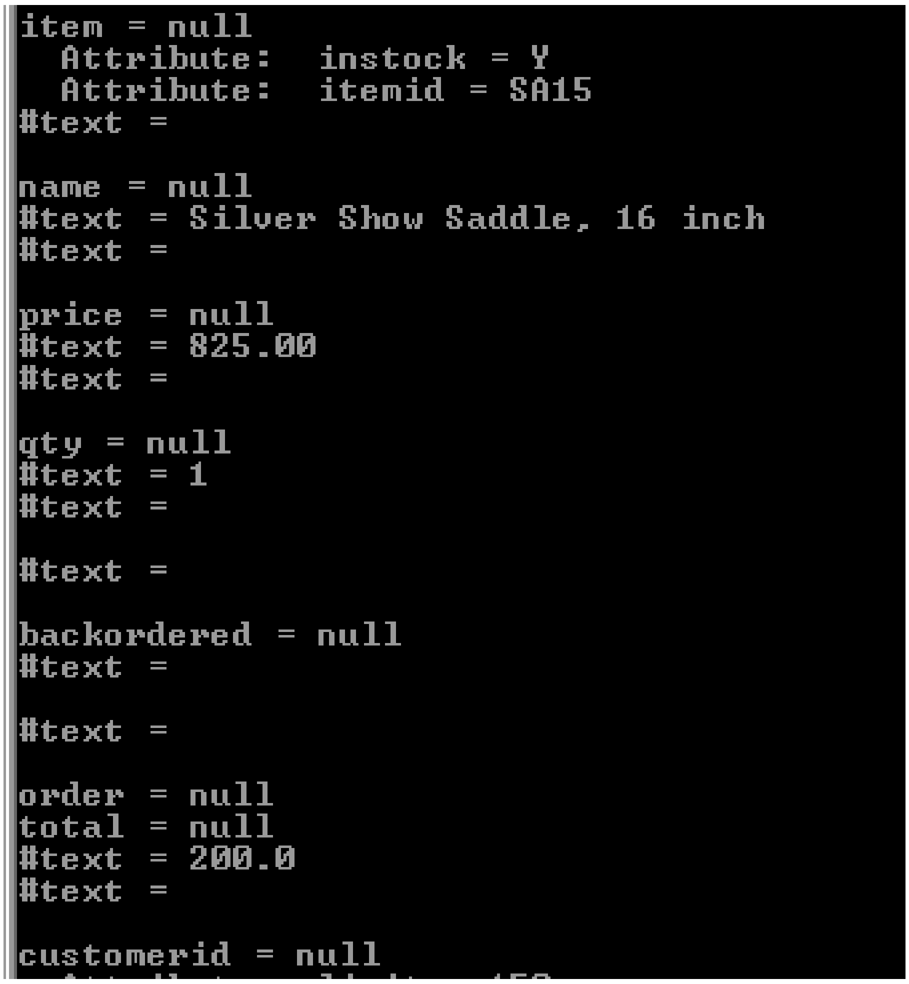
Notice that because no content was added to the element, it is an empty element. An empty element has no content, and can be written with a special shorthand:

```
<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 {
  ...
```



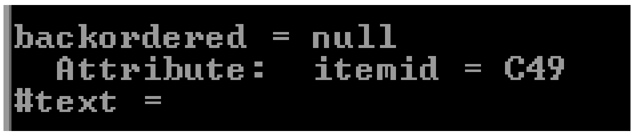
```
item = null
  Attribute:  instock = Y
  Attribute:  itemid = SA15
#text =

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, and then it 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 =
```

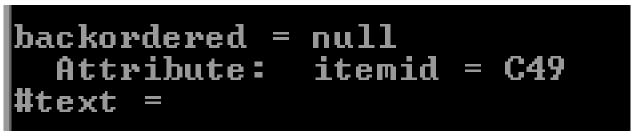
It's important to note that `setAttribute()` creates the attribute node if one by that name doesn't exist, so in this case the application could have skipped `createAttribute()` entirely.

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("customerid")
                              .item(0);
customer.removeAttribute("limit");

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



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

However, the next step utilizes the limit information, so you should remove this 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.

```
...

  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 (Exception e) {
      System.out.print("Problem parsing the file: "+e.getMessage());
    }
...
  thisOrder.insertBefore(totalElement, thisOrder.getFirstChild());
}


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());

...
```

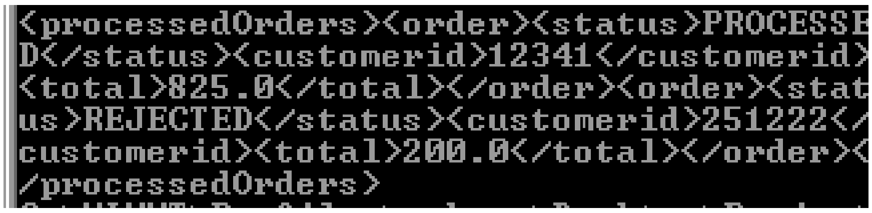
After processing `orders.xml`, the application creates a new element, `processedOrders`, which will eventually become the root element of the new document. It then runs through each of the orders. For each one, it extracts the `total` and `limit` information.

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`.



```
<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 will create extra white space (text) nodes.

```
...

 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.");
    }
    ...
```

Identity transformations

With a simple `Document` like those in this tutorial, it's easy to assume that outputting XML is simple, but remember to consider the many factors that complicate matters -- rare occurrences such as files whose content is affected by a DTD or schema. In most cases, it's better to rely on an application that already takes all of these possibilities into account.

One way that developers frequently choose to serialize their DOM `Document` s is to create an **identity transform**. This is an XSL Transformation that doesn't include a stylesheet. For example:

```
    ...

     import javax.xml.transform.TransformerFactory;
    import javax.xml.transform.Transformer;
    import javax.xml.transform.dom.DOMSource;
    import javax.xml.transform.stream.StreamResult;
    import java.io.FileOutputStream;
    ...
         newdoc.appendChild(newRoot);

     try {
          DOMSource source = new DOMSource(newdoc);
          StreamResult result =
                    new StreamResult(new FileOutputStream("processed.xml"));

          TransformerFactory transFactory = TransformerFactory.newInstance();
          Transformer transformer = transFactory.newTransformer();

          transformer.transform(source, result);
       } catch (Exception e){
          e.printStackTrace();
       }
      }
    }
```

Here, you are creating a source and a result, but because you're working with an identity transformation, you're not creating an object to represent the stylesheet. If this were an actual transformation, the stylesheet would be used in the creation of the `Transformer`. Instead, the `Transformer` simply takes the source (the

`Document`) and sends it to the result (the `processed.xml` file).

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 has covered the concepts behind the DOM and illustrated them with examples in Java code. Implementations of the DOM are also available in C++, Perl, and other languages.

Resources

Learn

- For a basic grounding in XML read through the "Introduction to XML" tutorial (*developerWorks*, August 2002).
- Check out the W3C DOM page for DOM Recommendations for Level 1, Level 2, and Level 3.
- In Brett McLaughlin's tip "Moving DOM nodes," learn how to move nodes from one document to another without encountering a common exception (*developerWorks*, March 2001).
- Try Brett McLaughlin's tip "Converting from DOM" which explains how to turn a DOM `Document` into a SAX stream or JDOM document (*developerWorks*, April 2001).
- Read Nicholas Chase's tip "Traversing an XML document with a `TreeWalker`" (*developerWorks*, October 2002) for an alternate way of looking at a document using DOM Level 2's `Traversal` module, and his tip "Using a DOM `NodeFilter`" to expand on that capability (*developerWorks*, November 2002).
- One alternative to the DOM is the Simple API for XML, or SAX. SAX allows you to process a document as it's being read, which avoids the need to wait for all of it to be stored before taking action. Find out more about SAX in the *developerWorks* tutorial "Understanding SAX" (updated July 2003).
- Order *XML Primer Plus*, by Nicholas Chase, the author of this tutorial (http://www.amazon.com/exec/obidos/ASIN/0672324229/ref=nosim/thevanguardsc-20).
- Find out how you can become an IBM Certified Developer in XML and related technologies (http://www-1.ibm.com/certify/certs/adcdxmlrt.shtml).

Get products and technologies

- Download the Java 2 SDK, Standard Edition version 1.4.2 (http://java.sun.com/j2se/1.4.2/download.html).
- If you're using an older version of Java, download the Apache project's Xerces-Java (http://xml.apache.org/xerces2-j/index.html), or Sun's Java API for XML Parsing (JAXP), part of the Java Web Services Developer Pack (http://java.sun.com/webservices/downloads/webservicespack.html).
- Download Xerces C++, a validating DOM parser (http://xml.apache.org/xerces-c/index.html).
- Download Xerces.pm, a Perl API implemented using the Xerces C++ API, which provides access to most of the C++ API from Perl (http://xml.apache.org/xerces-p/index.html).
- Get IBM's DB2 database (http://www.ibm.com/software/data/db2/) which provides not only relational database storage, but also XML-related tools such as the DB2 XML Extender

(http://www.ibm.com/software/data/db2/extenders/xmlext/) which provides a bridge between XML and relational systems. Visit the DB2 content area to learn more about DB2 (http://www.ibm.com/developerworks/db2).

About the author

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