Lowering the bar of the DOM API
A few easy steps to begin accessing XML data in Java

Bruce Martin
jGuru
March 2000

XML is a popular way to represent data in a portable, vendor-neutral, readable format. The Document Object Model (DOM) is an application programmer's interface to XML data. Unfortunately, the DOM is a fairly complex API with a high learning curve. But if you know the DTD of the data you are accessing, it's not too difficult. This article illustrates a few easy steps to begin accessing XML data using the DOM in Java.

Contents:

The Extended Markup Language, or XML, has gained widespread popularity as a way to represent data in a portable, vendor-neutral, readable format. Many software vendors have announced "support for XML," usually meaning that their software products will produce or consume XML data.

XML is also being viewed as the lingua franca for data exchange between enterprises. It allows enterprises to agree upon XML Document Type Definitions, or DTDs, for the data being exchanged. These DTDs are independent of the database schema used by the enterprises.

Standards groups representing almost every human endeavor are agreeing upon DTDs for exchanging data. One of many examples is the International Press Telecommunications Council (see Resources); it has defined an XML DTD allowing "news information to be transferred with markup and be easily transformed into an electronically publishable format." These vertical market standards will allow diverse applications to exchange data in unforeseen ways.

The XML specification (see Resources) defined by the W3C defines the syntax and semantics of the language. In order to process XML, an XML document needs to be parsed. It would be undesirable if every program that needed to process XML in some way had to parse an XML document, given that the syntax and semantics of the language are complex. To solve this problem, the W3C has defined the Document Object Model (DOM) (see Resources). The DOM is an application programmer's interface to XML data. Most XML parsers produce a DOM representation of the parsed XML.

The DOM standard
The DOM API is specified as a series of CORBA IDL interfaces (see Resources). It represents a parsed XML document as an abstract tree. It is abstract because only the interfaces reflect a tree structure. The actual data structures and algorithms implementing the abstract tree need not be a tree.

Since the DOM API is specified in CORBA IDL, it is available from many programming languages, including the Java programming language. We assume the standard Java language binding in this article. The DOM specification gives the Java binding API in detail.

The DOM Level 1 specification was accepted in 1998. It left several areas unspecified in order to learn from implementation experience. The DOM Level 2 augments Level 1 with support for XML name spaces, document creation, views, style sheets, etc. The Level 2 specification is open for public comment. Technically, it has not yet been finalized but it is fairly stable.

There are many XML parsers available for Java programs capable of producing DOM Level 1 representations of an XML document. Therefore, the code given here only assumes the DOM Level 1 subset.

Generic versus DTD-specific code
Java code that uses the DOM API is either *generic* or *DTD-specific*. Generic code will operate correctly for any XML document. Generic code is more difficult to write because it must generically navigate the DOM tree, considering all possibilities. The code cannot depend on any particular element, attribute or document structure. Generic code accomplishes generic tasks, such as checking the spelling of the text in the document, counting the number of words, sending the document across a network, and so on.

DTD-specific code, on the other hand, is written with a particular DTD in mind. It will fail to operate on an XML document that is defined by a different DTD. DTD-specific code is easier to write because it assumes that the XML document has the format specified by the particular DTD. For example, if a DTD declares that an element named "NAME" has a required attribute named "GIVEN", the Java code can assume that the attribute is present and simply use the DOM `getAttribute()` call.

This article helps you to write DTD-specific Java code. Writing generic code is a task to tackle after learning how to write DTD-specific code.

An example
In order to illustrate using the DOM API from a Java program, we use a purchase order since it is a typical business-to-business application of XML and has a fairly rich XML structure.

Here is the DTD for the purchase order that we used.

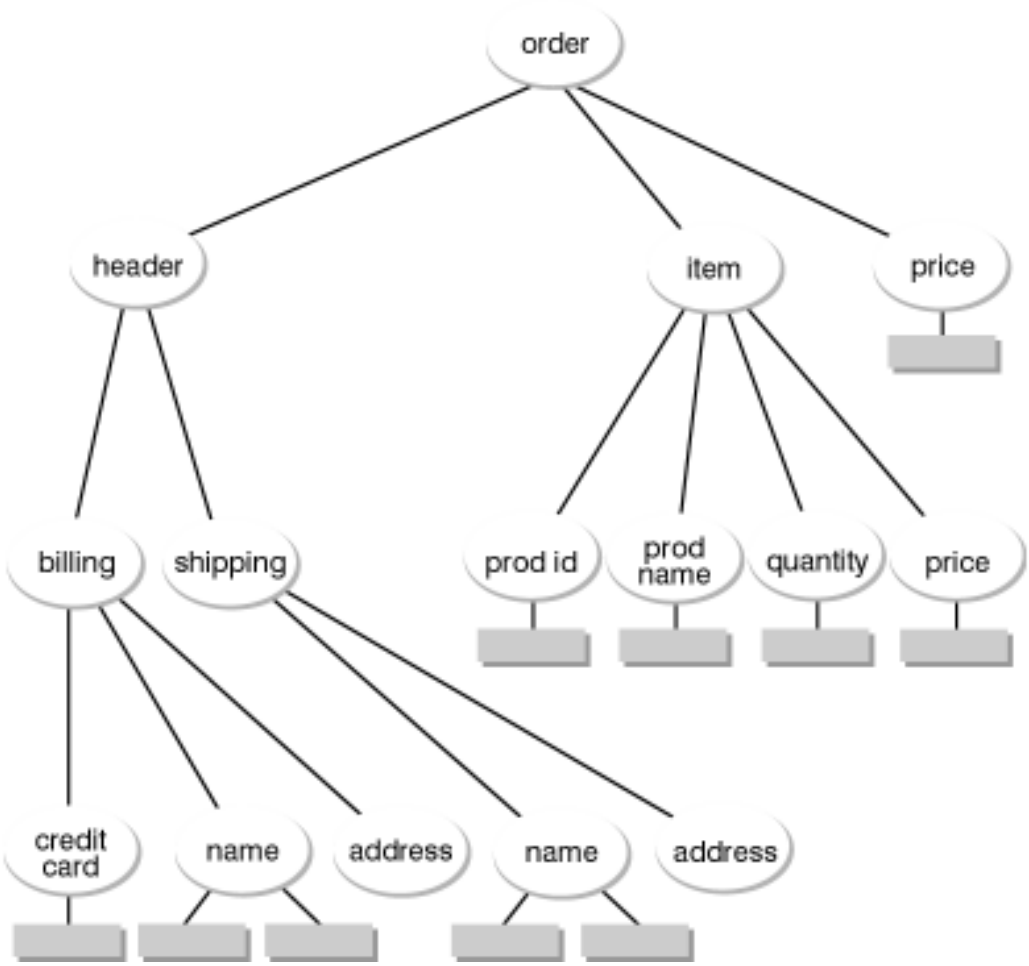
```
<?xml encoding="US-ASCII"?>
<!ELEMENT order (header,item+,price)>
<!ELEMENT header (billing,shipping)>
<!ELEMENT billing (name,address,creditCard)>
<!ELEMENT shipping (name,address)>
<!ELEMENT name EMPTY>
<!ATTLIST name
  given   CDATA #REQUIRED
  family  CDATA #REQUIRED
  >
<!ELEMENT address (street,city,state,zipcode,country,phone)>
<!ELEMENT item (prodId,prodName,quantity,price)>
<!ELEMENT creditCard (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT prodId (#PCDATA)>
<!ELEMENT prodName (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Here's an XML document with a purchase order of 100 widgets:

```
<?xml version="1.0"?>
<!DOCTYPE order SYSTEM "order.dtd">
<order>
        <header>
                <billing>
                        <name given="Jane" family="Doe"/>
                        <address>
                                <street>555 Main Street</street>
                                <city>Mill Valley</city>
                                <state>California</state>
                                <zipcode>94520</zipcode>
                                <country>USA</country>
                                <phone>707 555-1000</phone>
                        </address>
                        <creditCard>4555 5555 5555 5555</creditCard>
                </billing>
                <shipping>
                        <name given="John" family="Doe"/>
                        <address>
                                <street>100 Main Street</street>
                                <city>Brisbane</city>
                                <state>California</state>
                                <zipcode>94005</zipcode>
                                <country>USA</country>
                                <phone>415 555-9999</phone>
                        </address>
                </shipping>
        </header>
        <item>
                <prodId>5555555</prodId>
                <prodName>Widget</prodName>
                <quantity>100</quantity>
```

```
                <price>.25</price>
        </item>
        <price>25.00</price>
</order>
```

XML parsers *abstractly* represent the above XML document as a tree. Graphically that tree would look like this:



The ovals represent XML elements. The boxes represent the data. The lines from the `name` elements represent XML attributes. The `address` element is not shown graphically in detail.

A few simple steps to access XML data in a Java program
We now illustrate an important part of the DOM API with DTD-specific Java code. In particular, we illustrate the following DOM operations:

- `getDocType`
- `getName`
- `getElementsByTagName`
- `item`
- `getFirstChild`
- `getNodeValue`
- `getAttribute`
- `getChildNodes`
- `getLength`
- `getTagName`

In the sample code given in this article, all underlined method calls are part of the DOM API. The complete source code is available for download.

Define a simple interface to access the DOM document
The first step is to define an *abstract* interface that greatly simplifies the code using the XML document. For our example DTD,

we define the following interface:

```
interface order {
    String creditCard();
    String billingName();
    double totalPrice();
    boolean authorizeCredit();
}
```

Code accessing the XML data simply calls the operations defined by the interface.

Note that there are many implementations of this interface, including ones that have nothing to do with XML. (An implementation could be making queries on a database, for example.) Of course, here we are interested in implementations that process XML data that meets the order DTD given previously using the DOM API.

Implement the interface
Now implement a class that implements the interface and wraps the DOM document. For the example, we define:

```
class orderImpl implements order {
    Document theDocument;
```

Bind the DOM document to the wrapper class in the constructor
Pass the DOM document to the constructor of the wrapper class. The constructor checks the type of the document to ensure that the document does indeed conform to the order DTD. Recall that the code is specific to this DTD, and makes assumptions about the content and structure of the data.

```
public orderImpl(Document document) throws Exception {
  theDocument = document;
  DocumentType docType = theDocument.getDoctype();
  if (docType==null) throw new Exception("Cannot determine document type.");
  if (!docType.getName().equals("order")) throw new ¬
     Exception("Document is not an order.");
}
```

Note that the code gets the type of the document using the `getDoctype` operation defined by the DOM on the `Document` interface. The `getDoctype` operation returns an object supporting the `DocumentType` interface. Its name represents the type of the document.

Also note that some DOM implementations return `null` for the `getDoctype` operation and thus cannot be used with this constructor.

After providing code to bind the document to the wrapper class, implement the interface. Each of the operations in our example interface illustrates accomplishing particular tasks with the DOM.

Returning the value of a unique element
The `creditCard` method illustrates returning the value of a particular element. It uses the `getElementsByTagName` operation defined on the `Document` interface.

```
public String creditCard() {
    NodeList nl = theDocument.getElementsByTagName("creditCard");
    return nl.item(0).getFirstChild().getNodeValue();
}
```

In general, `getElementsByTagName` returns a list of elements. By examining the DTD for our example we know that a list with a single element will be returned because there is only a single element named "creditCard". That single element is available as `item(0)`. Thus, `nl.item(0)` is graphically represented as:

The string value of the `creditCard` element is obtained by calling `getFirstChild().getNodeValue()` on the credit card node.

Note that the `getElementsByTagName(elementName)` operation returns *all* elements in the document named by `elementName`. It is defined to return the elements in a preorder traversal of the document tree.

Since in our example the element name `creditCard` is unique, finding the element was straightforward. However, other elements, such as `name`, are not unique. We cannot just use the first element returned by `getElementsByTagName`. In fact, there is a `name` element that is a sub-element of the `billing` element and there is a `name` element that is a sub-element of the `shipping` element.

Returning the value of an attribute of an element with a unique subtree
The `billingName` method shows an approach to dealing with finding the value of non-unique elements, such as `name`. Note that `name` is not unique within the entire document but it is unique within the subtree rooted at the `billing` subtree. Also, note that the `billing` element is unique within the entire document. Thus, we can simply call `getElementsByTagName("billing")` on the entire document and then call `getElementsByTagName` on the returned `billing` element. This is possible because `getElementsByTagName` is also defined by the `Element` interface in the DOM API.

```
public String billingName() {
    NodeList bl = theDocument.getElementsByTagName("billing");
    NodeList nl = ((Element)bl.item(0)).getElementsByTagName("name");
    Element name = (Element)nl.item(0);
    return name.getAttribute("given")+" "+name.getAttribute("family");
}
```

Another technique illustrated by the `billingName` method is to obtain the value of an attribute from an element. Note from our DTD that the `name` element is defined to have two attributes: *given* and *family*. The `getAttribute` operation defined by the `Element` interface returns the text value of the attribute.

Returning the value of an element without a unique subtree
Now consider the `price` element. We cannot use the previous technique since `price` is a sub-element of the document and of each `item` element. The `totalPrice` method illustrates another approach to finding the value of non-unique elements. We depend on the document structure here, that is, we know we want the top level `price` element.

```
public double totalPrice() {
    NodeList nl=theDocument.getDocumentElement().getChildNodes();
    Element candidateElement=null;
    for (int i=0; i<nl.getLength(); i++) {
      if (nl.item(i) instanceof Element) {
          candidateElement = (Element)nl.item(i);
          if (candidateElement.getTagName().equals("price")) break;
              }
          }
          return Double.parseDouble(candidateElement.getFirstChild().getNodeValue());
}
```

The `getDocumentElement` operation returns an element representing the document. From there, we get its children nodes using `getChildNodes`. By examining the DTD we see that the children elements consist of a single `header` element, at least one `item` element, and a `price` element. So we simply loop through the children nodes until we find one with a tag name of `price`.

Again, note that once we have the element, we get its value by calling `getFirstChild().getNodeValue()`.

Abstraction
The `creditCard`, `billingName`, and `totalPrice` operations of our interface are *basic*; they simply find and return the corresponding XML elements. On the other hand, our interface also contains the abstract `authorizeCredit` method. There is no corresponding element in the XML.

The implementation of `authorizeCredit` is given below. It simply uses the `billingName`, `creditCard`, and `totalPrice` methods we have implemented in the wrapper class.

```
    public boolean authorizeCredit() {
        // illustrates abstraction
        return authorize(
                    this.billingName(),
                    this.creditCard(),
                    this.totalPrice());
    }
```

Client use of the class
We defined an abstract interface for accessing our XML document and we implemented all of its methods in a wrapper class using 10 important DOM operations. The only thing left is to show some Java code that uses the interface we have defined.

The code below simply invokes the parser, passes the DOM document returned by the parser to the constructor of our wrapper class, and then invokes each of the methods we implemented. This code uses the IBM XML Parser for Java (see Resources).

The use of other parsers would be similar.

```
import com.ibm.xml.xpk4j.xml4j2.*;
import java.io.*;

public class test {
    public static void main(String[] args) {
        XML4J2DOMSource parser = new XML4J2DOMSource();
        try  {
            parser.parse("order.xml");
            order theOrder = new orderImpl(parser.getDocument());
            System.out.println("The credit card is "+theOrder.creditCard());
            System.out.println("The total price is "+theOrder.totalPrice());
            System.out.println("The billing name is "+theOrder.billingName());
            theOrder.authorizeCredit()
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Our fairly simple example has shown 10 important operations of the DOM API. Using these operations we have illustrated how to find, navigate, iterate elements and obtain element and attribute values when the DTD is known. It is a solid start to learning the rest of the DOM API.

Resources

- **XML Parsers and DOM Implementations**
  If you are going to write Java code with the DOM API, you will need an implementation of the DOM and most likely a parser. Fortunately, there are several available today. Here are some sources:
  - Apache Xerces XML Parser
  - IBM's XML Parser for Java
  - Oracle (You must be a member to access the Oracle Technology Network. Registration is free.)
  - Sun's Java Project X
- Xbeans
  Xbeans is an open source repository of Java Beans that process DOM documents in fairly generic ways. Xbeans can be composed to create distributed applications that process XML. This is a good source of generic DOM code.
- jGuru XML FAQ
- An example of a standards group that is agreeing upon a DTD for exchanging data is the International Press Telecommunications Council.
- XML Specification
- DOM Specification
- CORBA IDL

About the author

Bruce Martin is one of the pioneers of distributed object computing. At Hewlett Packard Laboratories in the early 90s, he designed and implemented an interface definition language that became the basis for HP's original CORBA submission. At Sun Microsystems, he was one of Sun's CORBA architects and was the primary author of five of the OMG's CORBA Services specifications. At Inprise Corporation, Bruce was an architect and developer of Inprise's first CORBA-based Java Application Server. Bruce has extensive practical experience with Java, XML, and the DOM.

Bruce is now a software guru at jGuru. He is championing xbeans.org, an open-source project to create a repository of Java Beans that process XML and can be easily composed into distributed applications.

Bruce received Ph.D. and Masters degrees in Computer Science from the University of California at San Diego, and a Bachelors degree in Computer Science from the University of California at Berkeley.

What do you think of this article?

Killer! Good stuff So-so; not bad Needs work Lame!

Comments?

Submit feedback