When to Use DOM

The Document Object Model standard is, above all, designed for *documents* (for example, articles and books). In addition, the JAXP 1.2 implementation supports XML Schema, something that may be an important consideration for any given application.

On the other hand, if you are dealing with simple *data* structures and if XML Schema isn't a big part of your plans, then you may find that one of the more object-oriented standards, such as JDOM and dom4j, is better suited for your purpose.

From the start, DOM was intended to be language-neutral. Because it was designed for use with languages such as C and Perl, DOM does not take advantage of Java's object-oriented features. That fact, in addition to the distinction between documents and data, also helps to account for the ways in which processing a DOM differs from processing a JDOM or dom4j structure.

In this section, we'll examine the differences between the models underlying those standards to help you choose the one that is most appropriate for your application.

Documents Versus Data

The major point of departure between the document model used in DOM and the data model used in JDOM or dom4j lies in

- The kind of node that exists in the hierarchy
- The capacity for mixed content

It is the difference in what constitutes a "node" in the data hierarchy that primarily accounts for the differences in programming with these two models. However, the capacity for mixed content, more than anything else, accounts for the difference in how the standards define a node. So we start by examining DOM's mixed-content model.

Mixed-Content Model

Recall from the discussion of Documents and Data that text and elements can be freely intermixed in a DOM hierarchy. That kind of structure is dubbed *mixed content* in the DOM model.

Mixed content occurs frequently in documents. For example, suppose you wanted to represent this structure:

```
<sentence>This is an <bold>important</bold> idea.</sentence>
```

The hierarchy of DOM nodes would look something like this, where each line represents one node:

```
ELEMENT: sentence
  + TEXT: This is an
  + ELEMENT: bold
     + TEXT: important
  + TEXT: idea.
```

Note that the sentence element contains text, followed by a subelement, followed by additional text. It is the intermixing of text and elements that defines the mixed-content model.

Kinds of Nodes

To provide the capacity for mixed content, DOM nodes are inherently very simple. In the foregoing example, the "content" of the first element (its *value*) simply identifies the kind of node it is.

First-time users of a DOM are usually thrown by this fact. After navigating to the `<sentence>` node, they ask for the node's "content", and expect to get something useful. Instead, all they can find is the name of the element, `sentence`.

Note: The DOM Node API defines `nodeValue()`, `nodeType()`, and `nodeName()` methods. For the first element node, `nodeName()` returns `sentence`, while `nodeValue()` returns null. For the first text node, `nodeName()` returns `#text`, and `nodeValue()` returns `This is an .` The important point is that the *value* of an element is not the same as its *content*.

Instead, obtaining the content you care about when processing a DOM means inspecting the list of subelements the node contains, ignoring those you aren't interested in and processing the ones you do care about.

In our example, what does it mean if you ask for the "text" of the sentence? Any of the following could be reasonable, depending on your application:

- This is an
- This is an idea.
- This is an important idea.
- This is an `<bold>`important`</bold>` idea.

A Simpler Model

With DOM, you are free to create the semantics you need. However, you are also required to do the processing necessary to implement those semantics. Standards such as JDOM and dom4j, on the other hand, make it easier to do simple things, because each node in the hierarchy is an object.

Although JDOM and dom4j make allowances for elements having mixed content, they are not primarily designed for such situations. Instead, they are targeted for applications where the XML structure contains data.

As described in Documents and Data, the elements in a data structure typically contain either text or other elements, but not both. For example, here is some XML that represents a simple address book:

```
<addressbook>
  <entry>
    <name>Fred</name>
    <email>fred@home</email>
  </entry>
    ...
</addressbook>
```

Note: For very simple XML data structures like this one, you could also use the regular-expression package (`java.util.regex`) built into version 1.4 of the Java platform.

In JDOM and dom4j, after you navigate to an element that contains text, you invoke a method such as `text()` to get its content. When processing a DOM, though, you must inspect the list of subelements to "put together" the text of the node, as you saw earlier -- even if that list contains only one item (a `TEXT` node).

So for simple data structures such as the address book, you can save yourself a bit of work by using JDOM or dom4j. It may make sense to use one of those models even when the data is technically "mixed" but there is always one (and only one) segment of text for a given node.

Here is an example of that kind of structure, which would also be easily processed in JDOM or dom4j:

```
<addressbook>
  <entry>Fred
    <email>fred@home</email>
  </entry>
   ...
</addressbook>
```

Here, each entry has a bit of identifying text, followed by other elements. With this structure, the program could navigate to an entry, invoke `text()` to find out whom it belongs to, and process the `<email>` subelement if it is at the correct node.

Increasing the Complexity

But for you to get a full understanding of the kind of processing you need to do when searching or manipulating a DOM, it is important to know the kinds of nodes that a DOM can conceivably contain.

Here is an example that tries to bring the point home. It is a representation of this data:

```
<sentence>
  The &projectName; <![CDATA[<i>project</i>]]> is
  <?editor: red><bold>important</bold><?editor: normal>.
</sentence>
```

This sentence contains an *entity reference* -- a pointer to an entity that is defined elsewhere. In this case, the entity contains the name of the project. The example also contains a `CDATA` section (uninterpreted data, like `<pre>` data in HTML) as well as *processing instructions* (`<?...?>`), which in this case tell the editor which color to use when rendering the text.

Here is the DOM structure for that data. It's fairly representative of the kind of structure that a robust application should be prepared to handle:

```
+ ELEMENT: sentence
  + TEXT: The
  + ENTITY REF: projectName
    + COMMENT: The latest name we're using
    + TEXT: Eagle
  + CDATA: <i>project</i>
  + TEXT: is
  + PI: editor: red
  + ELEMENT: bold
    + TEXT: important
  + PI: editor: normal
```

This example depicts the kinds of nodes that may occur in a DOM. Although your application may be able to ignore most of them most of the time, a truly robust implementation needs to recognize and deal with each of them.

Similarly, the process of navigating to a node involves processing subelements--ignoring the ones you don't care about and inspecting the ones you do care about--until you find the node you are interested in.

A program that works on fixed, internally generated data can afford to make simplifying assumptions: that processing instructions, comments, `CDATA` nodes, and entity references will not exist in the data structure. But truly robust applications that work on a variety of data--especially data coming from the outside world--must be prepared to deal with all possible XML entities.

(A "simple" application will work only as long as the input data contains the simplified XML structures it expects. But there are no validation mechanisms to ensure that more complex structures will not exist. After all, XML was specifically designed to allow them.)

To be more robust, a DOM application must do these things:

1. When searching for an element:
    a. Ignore comments, attributes, and processing instructions.
    b. Allow for the possibility that subelements do not occur in the expected order.
    c. Skip over `TEXT` nodes that contain ignorable whitespace, if not validating.
2. When extracting text for a node:
    a. Extract text from `CDATA` nodes as well as text nodes.
    b. Ignore comments, attributes, and processing instructions when gathering the text.
    c. If an entity reference node or another element node is encountered, recurse (that is, apply the text-extraction procedure to all subnodes).

Note: The JAXP 1.2 parser does not insert entity reference nodes into the DOM. Instead, it inserts a `TEXT` node containing the contents of the reference. The JAXP 1.1 parser which is built into the 1.4 platform, on the other hand, does insert entity reference nodes. So a robust implementation that is parser-independent needs to be prepared to handle entity reference nodes.

Of course, many applications won't have to worry about such things, because the kind of data they see will be strictly controlled. But if the data can come from a variety of external sources, then the application will probably need to take these possibilities into account.

The code you need to carry out these functions is given near the end of the DOM tutorial in Searching for Nodes and Obtaining Node Content. Right now, the goal is simply to determine whether DOM is suitable for your application.

Choosing Your Model

As you can see, when you are using DOM, even a simple operation such as getting the text from a node can take a bit of programming. So if your programs handle simple data structures, then JDOM, dom4j, or even the 1.4 regular-expression package (`java.util.regex`) may be more appropriate for your needs.

For full-fledged documents and complex applications, on the other hand, DOM gives you a lot of flexibility. And if you need to use XML Schema, then again DOM is the way to go--for now, at least.

If you process both documents *and* data in the applications you develop, then DOM may still be your best choice. After all, after you have written the code to examine and process a DOM structure, it is fairly easy to customize it for a specific purpose. So choosing to do everything in DOM means that you'll only have to deal with one set of APIs, rather than two.

In addition, the DOM standard *is* a codified standard for an in-memory document model. It's powerful and robust, and it has many implementations. That is a significant decision-making factor for many large installations, particularly for large-scale applications that need to minimize costs resulting from API changes.

Finally, even though the text in an address book may not permit bold, italics, colors, and font sizes today, someday you may want to handle these things. Because DOM will handle virtually anything you throw at it, choosing DOM makes it easier to future proof your application.