Java makes the most of XML

Use Java to build applications that handle XML's extensibility

By Todd Sundsted
July 1999

Abstract
One of XML's advantages over HTML is its extensibility. This feature makes it possible to use XML to describe information in ways that would be impossible with HTML. This month, Todd demonstrates how to build a framework for processing XML in Java, aptly combining the inherent extensibility of both languages.

Introduction
Last month I presented my case for the place of XML in the enterprise (for last month's column, see Resources). I intentionally tried to look beyond the publishing aspects of XML to focus on application integration and data exchange issues. I demonstrated how easy it is to parse and validate XML using commonly available Java tools, and I compared those methods to more traditional ad hoc methods.

This month, I wish to carry the thread further--parsing and validating are fine as far as they go, but they don't go very far. The problem at hand typically involves doing something with the parsed information. But what if you don't understand the tags used to generate the information? Come walk with me a bit farther along the border between Java and XML, and I'll show you how to use Java to solve that problem, too.

XML tags: What to do?
Let's proceed straight to the heart of the matter. The feature of XML we need to address is its ability to define new tags. A tag in XML says something about the meaning of the content (and about the other tags) it associates with. Because the set of tags in XML is open (unlike HTML, where the set is closed), it's impossible to build an application that handles the entire tag set right out of the box. This introduces a bit of uncertainty into the process. What exactly do you do with tags you don't understand?

- **Applications can ignore novel tags.** This was the approach typically taken by browser vendors during the height of the browser wars. Leading browser vendors merrily defined new tags with each release of their product, and each browser distribution quietly ignored those tags it didn't understand. This approach is safe but not very satisfactory.
- **Organizations can standardize on a set of tags.** This approach cleverly sidesteps the entire problem. You define a set of standard tags and a document type definition (DTD), and then reject any XML that doesn't fit the mold. This is actually the right solution for many problems. Sales orders, for example, fit a well-defined pattern. Nothing is gained by allowing e-commerce partners to define new tags (at least without constraint--the case could be made for the applicability of certain well-constrained tag definitions, such as macros). Unfortunately, not all applications--XML browsers, for example--fit within this box.
- **Applications can try to figure out what to do with novel tags.** Browsers and content-presentation tools as well as other general-purpose XML tools must behave correctly in the presence of novel but valid tags. There are several ways to solve this problem. The Extensible Stylesheet Language (XSL) is one such

attempt. XSL provides a translation toolkit, which allows you to define a mapping or translation from a tag set you don't understand to a tag set you do understand (for example, XML to HTML). This solution, however, has its own limitations.

- **You can build a new framework.** While each of the solutions above has its place, we'll explore another solution altogether. Our solution calls for enabling the browser or XML tool to look for and download code designed to handle the novel tags and then integrate that code into the application. To do this, we'll build a new framework.

Before we can get down to the business of building our solution, we need to understand a little more about XML. In particular, we need to understand how to manipulate XML within an application. We need to understand the Document Object Model (DOM).

The DOM

The DOM is a platform-independent, programming-language-neutral API that allows programs to access and modify the content and the structure of XML documents from within applications.

At its core, the DOM defines a family of types that represent all the objects that make up an XML document: elements, attributes, entity references, comments, textual data, processing instructions, and the rest. (I use the word *object* throughout this article to loosely refer to the building-blocks of an XML document.) The DOM, originally envisioned as living inside a browser, has turned out to have a much broader impact. It is also worth noting that the DOM isn't specific to XML. It applies equally well to HTML.

To comprehend the DOM you need to remember that a key characteristic of XML is the notion that many documents can be represented as a hierarchical structure of content and markup. The code below, for example, represents a valid XML document:

```
<html>
 <head>
  <title>
   This is the title.
  </title>
 </head>
 <body>
  <h1>
   This is a headline.
  </h1>
  This is the body.
  And this is more of the body.
 </body>
</html>
```

I don't want to provide you with an XML primer, but I do want to make one point clear: A key requirement of XML (and HTML) is that tags must *nest*--they may not overlap. Therefore <one><two></two></one> counts as valid XML but <one><two></one></two> does not. As a consequence, well-formed XML documents map cleanly to a tree-like data structure. Next, we can transform the document above into the tree in Figure 1, below.

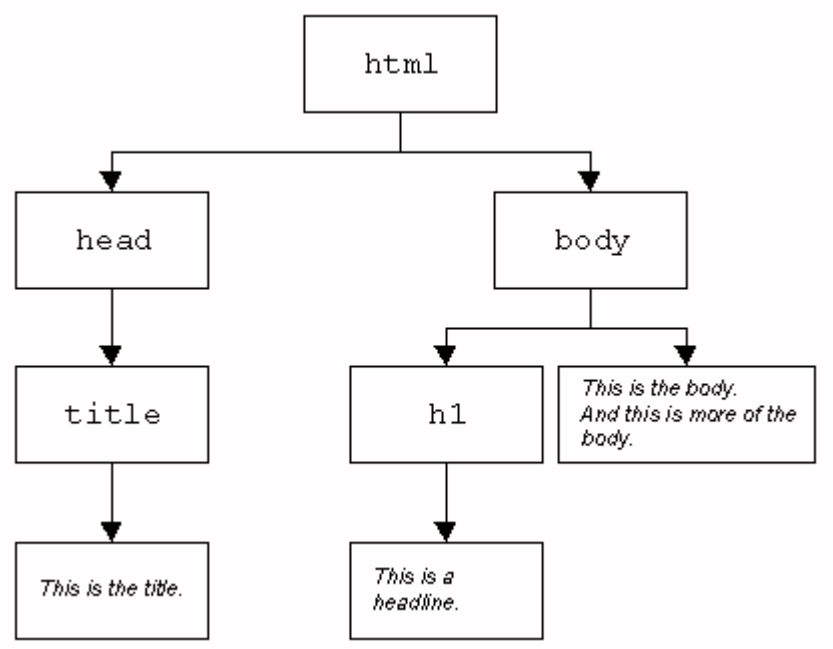

Figure 1. An XML tree

The DOM provides the mechanism we need to dynamically interact with the elements and content in an XML document. Consider the tree in Figure 1. I have mapped the tags (*elements,* in DOM parlance) that make up our initial XML document to the nodes of the tree in Figure 1.

Each tag has meaning within the context of the enclosing tags and the document as a whole. Consider the tags again. These tags clearly define presentation-related elements within a browser. As such, they have well-understood behavior associated with them. We expect the browser to know how to draw them within the browser window. The code that implements the behavior is present within the browser.

Now consider the code below, which represents an XML document with novel tags:

```
<mail>
<from>friend</from>
<to>friend</to>
<subject>GET RICH QUICK!!!</subject>
<body>
Dear Friend,
PLEASE READ THIS!!! It's easy to make money on the Internet. Just
follow this proven three-step plan.
</body>
</mail>
```

This is another small piece of XML. We clearly don't expect a browser to know what to do with these tags. To deal with them, the browser (or other general-purpose XML appliance) must be modified.

Figure 2, below, illustrates the general framework we'll employ to pull this off.

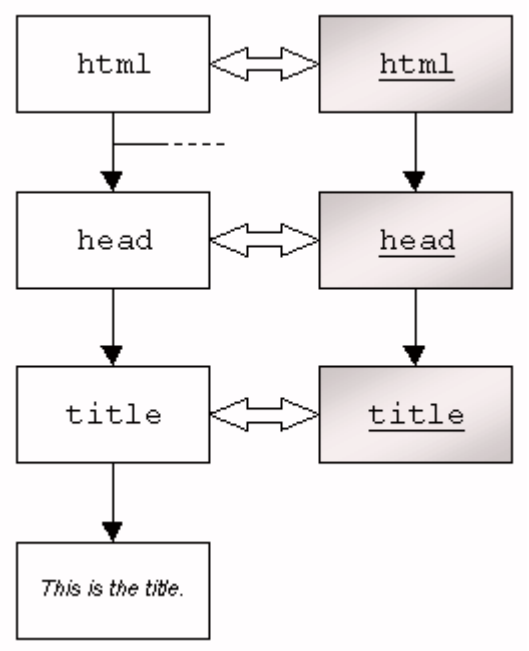

Figure 2. Framework for modifying the browser

In this example, each element of the DOM hierarchy on the left side maps to an element of the hierarchy on the right side. The DOM elements on the left represent the structure of the document. The elements on the right side represent the behavior of the structure elements. The behavior elements are arranged in a hierarchy as well, so that they can interact with each other in a manner that reflects the organization of the DOM model.

The Hook class

The building block of the behavior hierarchy is the Hook class. This class provides a behavioral "hook" into a behaviorless DOM tree. Here's the code for the Hook class:

```
import java.util.Vector;
import java.util.Enumeration;

import org.w3c.dom.Element;

public
class Hook {

  private
  Hook _hookParent = null;

  private
  Vector _vectorChildren = new Vector();

  private
  Element _element = null;

  public
  void
  setElement(Element element) {
    _element = element;
  }
```

```
  protected
  Element
  getElement() {
    return _element;
  }

  public
  void
  setParent(Hook hookParent) {
    _hookParent = hookParent;
  }

  protected
  Hook
  getParent() {
    return _hookParent;
  }

  public
  void
  addChild(Hook hookChild) {
    _vectorChildren.addElement(hookChild);
  }

  protected
  Enumeration
  getChildren() {
    return _vectorChildren.elements();
  }

  public
  Object
  build(Object object) {
    object = doOnNodeStart(object);
    Enumeration enumeration = _vectorChildren.elements();
    while (enumeration.hasMoreElements()) {
      Hook hook = (Hook)enumeration.nextElement();
      hook.build(object);
    }
    object = doOnNodeEnd(object);
    return object;
  }

  public
  Object
  doOnNodeStart(Object object) {
    return object;
  }

  public
  Object
  doOnNodeEnd(Object object) {
    return object;
  }
}
```

Hook is meant to be the supertype of a family of related subtypes. The Hook class itself doesn't implement any behavior. In fact, it doesn't implement any methods other than those needed to link parents and children. Families of subtypes should build on these primitives and define a collection of classes based around a common behavioral architecture.

The Filter class
The Filter class's sole method recursively builds the behavior tree from the DOM tree.

Let's take a look at the Filter class:

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

public
class Filter {

  public
  void
  filter(Hook hookParent, NodeList nodelist) {
    if (nodelist != null) {
      for (int i = 0; i < nodelist.getLength(); i++) {
        Node node = nodelist.item(i);
        // This handles what appears to be a bug in at least one
        // vendor's implementation (IBM's xml4j v. 2.0.6) of the
        // DOM. This bug seems to effect text nodes: the reported
        // length is nonzero but the returned list contains no valid
        // elements.
        if (node == null) break;
        if (node instanceof Element) {
          Element element = (Element)node;
          Hook hook = HookManager.createHook(element);
          filter(hook, element.getChildNodes());
          hook.setParent(hookParent);
          hookParent.addChild(hook);
        } else {
          filter(hookParent, node.getChildNodes());
        }
      }
    }
  }
}
```

The HookManager class
The HookManager class dynamically loads the code that implements the necessary behavior (the subtypes of class Hook).

Here's the code for the HookManager class:

```
import org.w3c.dom.Element;

public
class HookManager {

  public
  static
  Hook
  createHook(Element element) {
    Hook hook = null;
    try {
      hook = (Hook)Class.forName("_" + element.getTagName()).newInstance();
      hook.setElement(element);
    } catch (ClassNotFoundException exception) {
      hook = new Hook();
      hook.setElement(element);
    } catch (IllegalAccessException exception) {
      exception.printStackTrace(System.err);
    } catch (InstantiationException exception) {
      exception.printStackTrace(System.err);
    }
    return hook;
  }
}
```

An example

This example uses the framework described above to transform XML that *describes* a user interface into an actual user interface. First, here's the XML. I've included the DTD to further elucidate the relationship between elements:

```
<?xml version="1.0"?>

<!DOCTYPE GUI [

  <!ELEMENT GUI
    (FRAME)+
  >

  <!ELEMENT FRAME
    ( PANEL | BUTTON )*
  >
  <!ATTLIST FRAME
    LAYOUT ( Border | Flow ) "Border"
    TITLE CDATA "Frame"
  >

  <!ELEMENT PANEL
    ( PANEL | BUTTON )*
  >
  <!ATTLIST PANEL
    PLACEMENT ( North | South | East | West | Center | Default ) "Default"
    LAYOUT ( Border | Flow ) "Flow"
  >

  <!ELEMENT BUTTON
    EMPTY
  >
  <!ATTLIST BUTTON
    PLACEMENT ( North | South | East | West | Center | Default ) "Default"
    LABEL CDATA "Button"
```

```
  >
]>

<GUI>
 <FRAME TITLE="One">
  <PANEL PLACEMENT="North">
   <BUTTON LABEL="North1"/>
   <BUTTON LABEL="North2"/>
  </PANEL>
  <PANEL PLACEMENT="South">
   <BUTTON LABEL="South"/>
  </PANEL>
  <PANEL PLACEMENT="Center">
   <BUTTON LABEL="Center"/>
  </PANEL>
 </FRAME>
 <FRAME TITLE="Two">
  <PANEL PLACEMENT="North">
   <BUTTON LABEL="North"/>
  </PANEL>
  <PANEL PLACEMENT="South">
   <BUTTON LABEL="South"/>
  </PANEL>
  <PANEL PLACEMENT="Center" LAYOUT="Border">
   <PANEL PLACEMENT="North" LAYOUT="Border">
    <BUTTON PLACEMENT="North" LABEL="Center:North:North"/>
   </PANEL>
   <PANEL PLACEMENT="South" LAYOUT="Border">
    <BUTTON PLACEMENT="South" LABEL="Center:South:South"/>
   </PANEL>
   <PANEL PLACEMENT="Center" LAYOUT="Border">
    <BUTTON PLACEMENT="North" LABEL="Center:Center:North"/>
    <BUTTON PLACEMENT="South" LABEL="Center:Center:South"/>
    <BUTTON PLACEMENT="Center" LABEL="Center:Center:Center"/>
   </PANEL>
  </PANEL>
 </FRAME>
 <FRAME TITLE="Three" LAYOUT="Flow">
  <BUTTON LABEL="A"/>
  <BUTTON LABEL="B"/>
 </FRAME>
</GUI>
```

The source code and compiled class files are included in the jar file howto.jar. The sample XML file is test.xml. To use the software, you also need a copy of IBM's XML4J library, available in jar format as xml4j.jar. I'm distributing XML4J under IBM's XML4J commercial license. (See Resources to download the complete source code as a zip file.)

Follow the steps below to run the software:

1. Download the above files
2. Add both howto.jar and xml4j.jar to your CLASSPATH
3. From the command line, type: java Main test.xml

The application will read the specified XML file, parse and validate it, dynamically load and create the necessary hooks, and create the user interface.

Conclusion
When it comes to dynamically extending an application, Java's ability to dynamically load classes is quite a boon. It

makes frameworks like the one I presented above possible and truly expands the potential of a language like XML. Based on the framework and the sample application I presented, you can see how easy it would be to use XML as a scripted user interface definition language, or as the basis for a configuration tool, or any of a number of other possibilities. If you come up with a novel application, write me and tell me about it.

Next month, I'll continue my exploration of the compatibility of XML and Java. I'll demonstrate how to integrate a scripting language into the framework we've created and how to dynamically modify the DOM tree itself.

About the author

Todd Sundsted has been writing programs since computers became available in convenient desktop models. Though originally interested in building distributed object applications in C++, Todd moved on to the Java programming language when it became the obvious choice for that sort of thing. In addition to writing, Todd is an architect with ComFrame Software Corporation.

Resources

- Download the source code for this article in zip format: http://www.javaworld.com/jw-07-1999/howto/jw-07-howto.zip
- To download the source code in jar format, link to: http://www.javaworld.com/jw-07-1999/howto/howto.jar
- To download the test.xml file, link to: http://www.javaworld.com/jw-07-1999/howto/test.xml
- For IBM's xml4j.jar file, link to: http://www.javaworld.com/jw-07-1999/howto/xml4j.jar
- The W3C XML Web page. The source of the XML specification and related specifications: http://www.w3.org/xml
- The W3C DOM Web-page. The source of the DOM specification: http://www.w3.org/dom
- See IBM's alphaWorks site for the useful XML Parser toolkit: http://www.alphaworks.ibm.com/tech/xml4j

Related reading

- "XML and Java tackle enterprise application integration," Todd Sundsted (*JavaWorld,* June 1999). The first of Todd's **How-To Java** columns on the topic of XML and Java: http://www.javaworld.com/javaworld/jw-06-1999/jw-06-howto.html
- *XML.COM*--an online magazine dedicated to XML: http://www.xml.com/