JavaBeans by Mark Johnson

XML JavaBeans, Part 1

Make JavaBeans mobile and interoperable with XML

Summary
The buzziest of buzz words these days is XML, the acronym for the eXtensible Markup Language. But it's more than buzz. This rapidly developing technology is well-suited for use with JavaBeans -- and nicely complements Java, whose write-once, run-anywhere philosophy has given the world unprecedented network mobility. This article, the first of a two-part series, describes just one possible application for XML: making JavaBeans mobile and interoperable by representing them as XML documents. Follow along as columnist Mark Johnson describes XML, defines his own custom markup language, and creates a class that converts XML files to JavaBeans running in memory. *(4,200 words)*

HTML (Hypertext Markup Language) currently is *the* document format of the World Wide Web. Lately, though, there's been a lot of noise about XML (eXtensible Markup Language), which allows, among other things, the ability to define new markup tags (the bits between <angle brackets>), or even whole new markup languages. Some pundits even claim that XML may supplant HTML as the dominant information format on the Web.

Mail this article to a friend

For some, XML seems one of those ideas that, while exciting at first, isn't entirely useable in practice. How would a developer use XML in a real life system? What good is the ability to define custom tags if no browsers understand them? In this month's column, we'll look at a possible application of XML -- namely, using it as a serialization format for JavaBeans.

First, you'll read a quick rundown of what XML is and why so many people are so excited about it. Next, you'll hear about the World Wide Web Consortium's (W3C's) Document Object Model, the proposed standard for representing documents as data structures. As an example of processing a document as a data structure, we'll describe a very small custom markup language, and then implement a class that reads an XML file and transforms it into a JavaBean.

Please note that the primary purpose of this article is to provide an example of XML in use. While it is not an introduction to XML for the complete novice, this article should be comprehensible with just a bit of preparatory reading (see the introductory articles listed in the Resources section.)

What's wrong with HTML, anyway?

There's a great deal of introductory material on the Web about XML, so we're going to go over XML basics pretty quickly. Let's start by discussing why XML is necessary in the first place.

It's easy to make the argument that HTML enabled the explosion of the Web. Among the many strengths that have made HTML the dominant format for Web documents are the following:

- HTML is very easy to learn and use. Practically anyone with a pulse can learn to write HTML. Reading HTML in a Web browser is so simple and intuitive that just about everyone grasps it instantly.

- Logical layout makes HTML documents portable. HTML markup describes to a browser what roles various pieces of text play in a document (title, list element, and so on,) and the browser is free to decide how (or if) to display them. This provides a great deal of device independence.

- Hypertext forms webs of knowledge. One of the most useful features of HTML for many applications is the ability to make information "come alive" and refer to other information.

- HTML forms a framework for composite documents. The addition of applets and other sorts of "active" page elements provides immense creative control to developers on the Web "platform."

Despite these and the many other strengths that make HTML so useful and, well, *cool,* it has some serious drawbacks that are rapidly becoming obstacles to using it in serious data applications:

- HTML is a rapidly growing monster. It was originally designed for sharing documents between scientists at CERN. (CERN stands for *Conseil Europeen pour la Recherche Nucleaire,* the Center for European Nuclear Research, though its Web site consistently describes it as the European Laboratory for Particle Physics.) They wanted structured text with some simple outline capabilities, simple hyperlinks, primitive font control, and maybe some pretty pictures and colors, and that's what they created. It was simple, elegant, and useful. It's still useful, but simple and elegant have gone out the window as developers have demanded, and browser creators have developed, new features for HTML. The HTML specification has ballooned to enormous size with the addition of such features as scripting, frames, layers, tables, forms, style sheets, objects, applets, and on and on.

- HTML is set in stone. Within a particular version of the HTML standard, only certain tags, such as <TITLE> or <B> (for **boldface**), are recognizable HTML tags. If you're working in HTML, you're stuck with the tags recognized by the HTML spec (or your particular browser). If you want to define your own tags for some reason, you're out of luck.

- HTML is very browser-centric. HTML documents are, by and large, plain text with markup to provide display organization, some font control, and graphic content. They are documents written for humans to read, not for client-side programs to analyze and present. Because of this, HTML is not a good choice as an information format for automated data processing systems.

HTML mostly addresses presentation, not content. Generally, HTML tags describe how or in what context to display a particular piece of text. The semantics of the text, that is, what that text actually *means,* is lost in HTML.

What do the data mean?

This last deficiency of HTML is the clincher. As data become more mobile in data processing systems, it's necessary to transfer both the information and metainformation about what the data *mean.* A number in an HTML table may or may not reliably mean something when the document is read by a program. An XML document can be designed to express not simply how to display the data, but what data *mean.*

For example, an HTML table can display statistics for an individual baseball player, as in Figure 1.

HTML Source

```
<TABLE>
<TR ALIGN="CENTER" VALIGN="BOTTOM" BGCOLOR="#008080">
<TD><B>NO.</B></TD>
<TD><B>PLAYER</B></TD>
<TD><B>High School</B></TD>
<TD><B>AB</B></TD>
<TD><B>R</B></TD>
... (and so on) ...
```

Resulting table

NO.	PLAYER	High School	AB	R	H	HR	RBI	AVG
12	Jonas Grumby	Eaton	69	31	30	2	15	.435

Figure 1. Batting averages in an HTML <TABLE>

A row-column representation of these data is fine if what's needed is simply a static display of data in this particular format, but it's not a great representation if you want to associate meaning with the data in your application. Try writing a program that reads the HTML above, retrieves the information about, say, the hitter's runs-batted-in, and then does something with that quantity. With HTML, that's not easy to do in a general way. Imagine, though, that your data file looked something like what appears in Figure 2:

```
<?xml version="1.0">
<Player>
  <Name><First>Jonas</First><Last>Grumby</Last>&
lt;/Name>
  <Number>12</Number>
  <HighSchool>Eaton</HighSchool>
  <Stats">
    <Year>1997</Year>
    <AtBats>69</AtBats>
    <Runs>31</Runs>
```

```
    <Hits>30</Hits>
    <HomeRuns>2</HomeRuns>
    <RunsBattedIn>15</RunsBattedIn>
  </Stats>
</Player>
```

Figure 2. The batting information specified in XML

Figure 2 is a sample of XML that represents the same information as in Figure 1. It would be easy to pick out the "runs-batted-in" statistic in this document. The document could change structure radically, and the <RunsBattedIn> tag would still be relatively easy to find. The XML code in Figure 2 contains the same information as the HTML code in Figure 1, but it's represented in a way that indicates what the data mean, not just how to present the data.

Just as in HTML, a style sheet can be associated with XML, though XML's style language, XSL, is more powerful and cryptic than HTML's Cascading Style Sheets. In fact, XSL can convert XML into HTML for display by a browser! The XML above could be displayed in a browser just as it appears in Figure 1, but client-side programs could also collect and use such statistics, since there's an indication (via the tag) of what the data mean.

You may be wondering how I knew what tags to use in creating my sample XML file. Where did the tag names (like `RunsBattedIn`) come from? The answer is: I made them up. I just invented markup tags for my application out of thin air! Creating a new markup language is just like creating any other kind of custom file format. A developer simply creates a file format that meets the needs of the application. XML files are special in that they conform to the XML definition, and so programs that process them can expect input of a certain structure, and can reasonably reject inputs that don't follow that structure.

In the example above, I've created a new XML sublanguage simply by inventing new tags and using them consistently. XML also provides the option of specifying a *Document Type Definition* (DTD), which is a specification of what elements form a valid document. A DTD gives a developer much more control over the format of an XML document with a DTD than without one. We're not going to cover DTDs in this article, but they are a core XML concept.

If you think XML looks like HTML, it's because they're close cousins. Both XML and HTML are applications of SGML (Standard Generalized Markup Language), which is a *metalanguage* -- that is, a language for describing languages. SGML is an extremely powerful, flexible, and complex tool, and its complexity has led to its use primarily in huge organizations, like governments and large corporations. XML is a subset of SGML that retains most of SGML's power while simplifying it for use by common mortals. In fact, both HTML and XML are actually specified as DTDs in SGML. (Are you burned out on acronyms yet?)

Referring again to Figure 2, notice that the XML indicates what the data mean, not how they are to be displayed. Notice also that the tags certainly are not standard HTML. (Let's hope that the <RunsBattedIn> tag is never made part of the HTML standard!) This example shows one of the strengths of XML: the ability to define custom markup tags to suit a particular application. Finally, notice that the batting average doesn't appear in the XML. That's because the average could be calculated from the other values.

One of XML's most powerful abilities is that, with XML, a system designer can create a custom data markup language that maintains the semantics of the data.

Now that we've seen what an XML file looks like, let's find out how to process XML in Java.

The Document Object Model (DOM)

Languages like HTML and XML are often called *structured markup languages.* This means that the markup in a file has a particular structure that means something to the applications that process it. For example, HTML files start with a `<HEAD>` element, which contains a `<TITLE>`, which contains text, and so on. An HTML browser displays the information in an HTML file *based on this structure.*

The structure of XML and HTML is so simple and consistent that it's very easy to represent any XML or HTML document as a tree of objects, whether in Java or in some other programming language. The World Wide Web Consortium (W3C) has defined a complete set of objects to be used for processing XML and HTML documents as trees. The specification for this set of objects is called the *Document Object Model* (DOM; see the link to the DOM spec in Resources below.) Let's go back to our example and see how to represent a document as a tree.

Take another look at the XML in Figure 2. You'll notice that the document has a hierarchical structure: elements contain other elements. For example, the `<Name>` element contains a `<First>` element and a `<Last>` element. This "contains" relationship could be represented in a class diagram, as shown in Figure 3. In this diagram, a class `Player` contains an integer `Number` and a string `High School`, and also contains references to two other objects, a `PersonName` object and a `Statistics` object. Each of these objects contains other objects, as well.

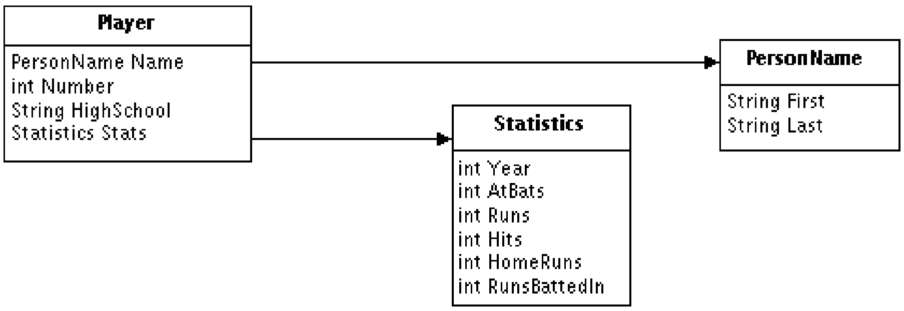

Figure 3. A class diagram for batting statistics

HTML documents have this "contains" structure, too, but in an HTML document, the structure reflects information about the document (the title, headings, and so on) instead of reflecting the structure of the *information* represented by the document. The XML document above reflects how information relates to other information.

In general, you can think of an XML (or HTML) document as a tree structure, with the "contains" relationship in the document corresponding to a parent-child relationship in the tree. One possible graphic representation of this idea appears in Figure 4. (The particular shapes I chose for the document aren't standard notation. They simply distinguish different object types.)

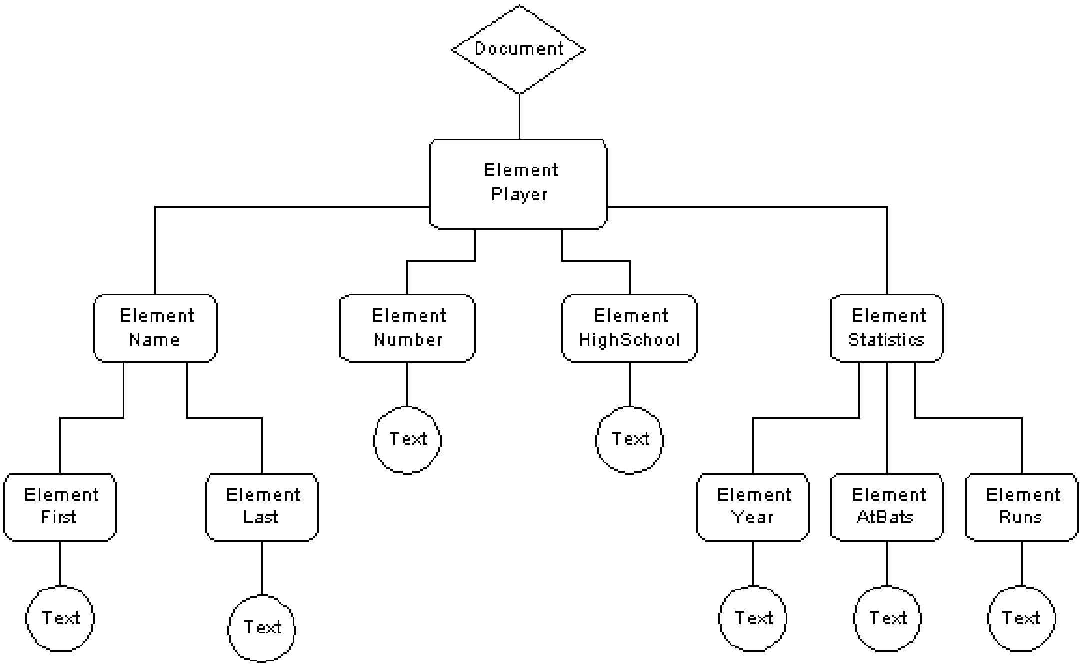

Figure 4. The baseball statistics document as a tree

So, an XML document can be represented by a program as a tree of `Element` objects, each of which may contain other `Element` objects and `Text` objects. The entire document is rooted in a single `Document` object. The `Text` objects contain the data for the object.

The Java package org.w3c.dom is the standard "binding" of the DOM specification in terms of Java interfaces (meaning all of their methods are abstract, and therefore have no implementation). Various vendors can implement the classes in this package in any way they wish. IBM, for example, has implemented this package in its *xml4j* distribution. There are several implementations of the DOM, most of which include vendor-specific extensions. We'll be using the DOM in the sample application below. See Resources to read up on the Document Object Model, and to find out how to get IBM's implementation of it.

DOM XML parsers

Now that we've defined a new language, and we have a data structure system (DOM) to represent any document in this language as a tree in memory, it would be useful to have a parser that automatically converts an XML document into its DOM tree. Once we have the DOM representation of a document in memory, we can process the document as a tree of nodes, instead of as a series of lines or tokens. What we need is a parser in Java which will read the XML file (which is only a text file, after all) and produce a single `Document` object containing a tree of DOM nodes that represent the document completely.

Fortunately, we don't have to go out and write a parser from scratch. Several companies and individuals have written parsers that read XML documents from files or streams and produce a DOM `Document` object, which is the root of a tree of DOM objects (as seen in Figure 4). The entire process of reading in an XML file and turning it into a usable tree is encapsulated in the parser. Many of the DOM object implementations also include extensions for going the other way; that is, a DOM `Document` can be printed as XML with a single method call.

In the sample code below (yes, we're finally getting to a coding example), I've used the parser from IBM's `xml4j` package, which is available free for noncommercial use from IBM's alphaWorks site (you can find the URL in Resources). IBM apparently has gone completely bananas for XML, and I consider alphaWorks to be one of the most interesting Java technology sites on the Web. The `xml4j` package implements the W3C DOM completely, extends it in a sensible yet encapsulated manner, and comes with copious, excellent documentation.

Beans as XML documents

In thinking about how to use XML with JavaBeans, I decided it would be interesting to use XML as a serialization format for beans. In other words, I decided to create a markup language that allows a user to create an XML file that specifies the values for a JavaBean's properties.

If you're not familiar with JavaBeans, the concepts of "properties" and "serialization" also may not be familiar to you. If this is the case, you may want to get some background by first reading some or all of the following articles:

- A Walking Tour of JavaBeans (JavaWorld, August 1997), a JavaBeans primer
- "Double Shot, Half Decaf, Skinny Latte" -- Customize your Java, a description of JavaBeans properties
- Do it the "Nescafé" way -- with freeze-dried JavaBeans, an introduction to serialization

If you're already familiar with these concepts (perhaps because you're a regular reader of the *JavaWorld* **JavaBeans** column,) just dive right in. You can always return to these references if there's something in this article you don't understand.

The example class we develop in this article is called `XMLBeanReader`. This class reads an XML file (of a specific format that we define) and uses its contents to create a JavaBean and initialize that bean's properties. The JavaBean class name and the property values all come from the XML file contents. All of the methods of this class are `static`.

If you understand how this small sample program works, you can extend it to handle hookup up event relationships, looking up default values for properties, or seeking out information that's not in the XML file itself in order to configure the JavaBean. The possibilities are endless, once you understand that a data structure of JavaBeans can be converted to an XML file and back.

The `XMLBeanReader` class works something like the standard Java serialization mechanism, in that it takes a "flat" stream of data and uses those data to set properties in a JavaBean. It doesn't create a new *class*. It uses XML to instantiate a JavaBean and set that bean's properties for a JavaBean class that already exists.

JavaBean Markup Language

Before writing any code, we need to define what our simple XML dialect looks like. Since our application deals with JavaBeans, I'm going to create a language that allows the user to specify a JavaBean and its class, and then specify a list of properties for the JavaBean.

Despite some similarities, the `XMLBeanReader` class you're about to see is in no way related to IBM's

Bean Markup Language (BML is available from the alphaWorks site -- see Resources for the appropriate link). Once you understand what's going on with the code in this column, though, you'll be better able to tackle projects using BML.

For this simple XML dialect, the only tags we need are:

- <JavaBean>: a tag enclosing a specification of the contents of a JavaBean
- <Properties>: a tag that encloses all <Property> elements of a particular JavaBean
- <>: a tag that encloses the value of a single property

Now, imagine we had a class `Player`, which was a JavaBean with four properties:

- `int Number`: the player's number
- `String HighSchool`: the name of the player's high school
- `PersonName Name`: a JavaBean of class `PersonName` that is the player's name
- `Statistics Stats`: a JavaBean of class `Statistics` containing player's batting statistics for a particular year

Notice that two of the property's values are JavaBeans themselves. The source code for the three JavaBean classes we'll be using can be found in Player.java, PersonName.java, and Statistics.java.

Given the JavaBean class we've just defined (`Player`), the tags we've just defined above, and the data from Figure 2, we could express the JavaBean in XML in terms of its properties, like this:

```
<JavaBean CLASS="Player">
  <Property NAME="Number">12</Property>
  <Property NAME="HighSchool">Eaton</Property>

  <!-- Notice that the value for the properties "Name" and
    -- "Stats" are themselves JavaBeans!
    -- Notice also that comments in XML files look
    -- just like comments in HTML files.
    -->
  <Property NAME="Name">
    <JavaBean CLASS="PersonName">
      <Properties>
        <Property NAME="First">Jonas</Property>
        <Property NAME="Last">Grumby</Property>
      <Properties>
    </JavaBean>
  </Property>

  <Property NAME="Stats">
    <JavaBean CLASS="Statistics">
      <Properties>
        <Property NAME="Year">1997</Property>
        <Property NAME="AtBats">69</Property>
        <Property NAME="Runs">31</Property>
```

```
        <Property NAME="Hits">30</Property>
        <Property NAME="HomeRuns">2</Property>
        <Property NAME="RunsBattedIn">15</Property>
      </Properties>
    </JavaBean>
  </Property>
</JavaBean>
```

Figure 5. Batting statistics represented in XML as a JavaBean

The first thing you'll notice in the code above is that there are strings embedded with quotes inside of the tags, like this:

```
<JavaBean CLASS="Player">
```

These strings are called *attributes,* and if you've written HTML, you've used them. They appear in tags such as hyperlinks (`<A HREF="...">`) and images (`<IMG SRC="...">`.) Attributes are simply another way of associating data with a DOM `Element` node. The method `Element.getAttribute()` returns an `Element`'s attribute with a specific name. There are no hard-and-fast rules about when to include data in an attribute and when to put it in the `Text` object inside the `Element`. I tend to use attributes for structural information (classes, property names, and so on) and `Text` node for instance data. Use what you consider the easiest.

Creating JavaBeans from XML

The `XMLBeanReader` class (source in XMLBeanReader.java) is quite straightforward. The `main()` method (lines 400 to 415) simply calls the static method `readXMLBean()`, passing it the name of the input file. The result returned from `readXMLBean()` is a JavaBean whose class and contents correspond to what was in the XML file. `main()` then checks to see if the JavaBean it just created has a method called `print()` and, if it does, `main()` invokes it. (Isn't reflection cool?)

`readXMLBean()` (lines 377 to 395) creates an XML parser and invokes it on the input file. The result of the parser's `readStream()` method is the document tree, which, if drawn, would look something like Figure 4. It then passes the top `Element` of the document tree (the tag of which must be **JavaBean**) to the static method `instantiateBean()`, which is where all the serious work is done. The result of `instantiateBean()` is the JavaBean that method created.

`instantiateBean()` (lines 269 to 294) creates a JavaBean from a DOM tree with a `<JavaBean>` element at the top. First, it creates an object of the type indicated by the `CLASS` attribute of the `<JavaBean>` element. It then finds the `<Properties>` node, which is a child of the top (`<JavaBean>`) node. Within this `<Properties>` may be several `<Property>` nodes, each of which corresponds to a property of the JavaBean. For each `<Property> Element` (among the children of `<Properties>`), `instantiateBean()` calls `setProperty()`, passing in the name of the property, the property descriptor (obtained by applying `java.beans.Introspector` to the created JavaBean), the `Element` corresponding to the property, and the new bean itself. When this whole process has completed, the JavaBean has been both created and initialized, and can be returned to the caller.

The `setProperty()` method (lines 048 to 249) takes care of setting a property of a JavaBean. It handles regular and indexed properties separately (by ignoring indexed properties, for the moment). In the (usual) case of a nonindexed property, `setProperty()` first searches all of the children of the `<Property>` for either a `Text` node or a `<JavaBean>` element, and records what it finds for later use. It also asks the `PropertyDescriptor` for the `setter` method for the property.

The remainder of `setProperty()` concerns itself with figuring out what arguments to pass to the `setter` method. There are a few possible conditions, all of which depend on the type of the property, or on the expected arguments of the `setter` method:

- **The property type is primitive.** (Lines 110 to 147.) Primitive types can all be constructed from a `String`, so if the type of the property is primitive (`boolean`, `byte`, `int`, `long`, `float`, or `double`), `setMethod` simply constructs an object of the appropriate type using the constructor that takes a single `String` as an argument and passing it the text value of the `Element`. Properties of primitive type `char` are handled as a special case, since I decided to encode them as their integer values, and they can't be readily constructed from strings. The object that the method constructs is used as the argument to the `setter` method.

- **A <JavaBean> Element appeared as a child of the <Property>, so the property must be a JavaBean.** (Lines 168 to 191.) In this case, the argument we want to pass to the setter method is actually a JavaBean. We need to instantiate that JavaBean before we can pass it to the setter, and we do so by *calling* `instantiateBean()` recursively, passing it the child `<JavaBean>` element. The resulting (JavaBean) object is placed in the argument list for the setter.

- **The setter method for the property takes as its only argument a String.** (Lines 193 to 203.) This is an easy case. The argument list for the setter contains simply the `Element` text of the `Property`.

- **The setter method for the property takes as its only argument an object that can be constructed from a String.** (Lines 205 to 222.) In this case, `setMethod()` does exactly what it did for the primitive case: it constructs an object of the appropriate type, using the `Element` text as the constructor argument, and then places this new object in the argument list (`setterArgs`) for the setter.

If none of these conditions are met, `setProperty()` isn't capable of setting the property, and it returns without doing anything. (It should probably throw an `IllegalArgumentException`, but this is just a demo program.)

Running the class file's `main()` method, and passing it the XML source file you see in Figure 5, produces a running JavaBean with all of its properties set to values that came (originally) from the XML, including properties whose values are JavaBeans. You can see the output of this program being run in Example.html. You see that the Statistics bean does indeed report its properties set correctly.

Conclusion

In a very small space, we've covered an enormous amount of ground. You've read about what XML is, and what it can do that HTML can't. The sample XML on batting statistics showed how the structure of XML can be used to reflect the meaning of the data, not its presentation. You then read about the standard programmatic interface to XML, the Document Object Model, and then saw an example of the DOM and XML in action, being used to create and initialize a new JavaBean instance.

Next month, we'll go the other way, creating a class that takes a running JavaBean and converts it to XML. We'll also use these two classes, `XMLBeanReader` and `XMLBeanWriter`, in a small application, to demonstrate the power and flexibility of XML.

Please write and let me know if you'd like to hear more about XML and JavaBeans; you can send e-mail using the address listed in my bio below. You can also send your comments on this article to *JavaWorld* by clicking the link at the bottom of the page. ■

Also this month in JavaWorld
Go

Resources

- For IBM's Bean Markup Language (BML), see
  http://www.alphaWorks.ibm.com/formula
- One of the better "one-stop shopping" sources for XML information is at XML.com. It has links to just about everything in the XML world. One of the more interesting things at this site is, believe it or not, the *commentary* on XML technology. See
  http://www.xml.com
- A current version of the *XML FAQ* by Peter Flynn, et al., appears at the following site. This is the version of the FAQ recommended by the W3C
  http://www.ucc.ie/xml/
- The parser from IBM's `xml4j` package is available free for noncommercial use. It's even free for commercial use, but be sure to read the license agreement first
  http://www.alphaWorks.ibm.com/formula/XML
- In a note unrelated to JavaBeans, but still too cool for words, check out Jikes, IBM's new *open source java compiler*! Find out about it at the alphaWorks site
  http://www.alphaWorks.ibm.com/formula/JikesOS
- For IBM's Bean Markup Language (BML), see
  http://www.alphaWorks.ibm.com/formula/BML
- If you're interested in the fine details of the current Document Object Model (Level 1) Specification, you can find it at the W3C's Web site
  http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html
- Microsoft has a good set of tutorials on XML
  http://www.microsoft.com/xml/tutorial/default.asp
- Microsoft also offers a whole XML "workshop" area. Don't try to access the workshop in Netscape, though: the table of contents doesn't work! These documents are free training, and are well-written (though the examples don't always work, even in IE5beta.) Just don't be fooled into thinking that everything there is open standard. Some of the tutorials and many of the articles are about Microsoft-only technology that won't work with all browsers or platforms.
  http://www.microsoft.com/xml/default.asp
- The source code for this article is available for download in Unix tar format
  http://www.javaworld.com/jw-02-1999/beans/XMLBeans.tar
- It's also available in zip format
  http://www.javaworld.com/jw-02-1999/beans/XMLBeans.zip
- You can also download a jar file with the class files, ready to run, from

http://www.javaworld.com/jw-02-1999/beans/XMLBeans.jar

About the author
Mark Johnson has a BS in Computer and Electrical Engineering from Purdue University (1986), and has been writing for *JavaWorld* since August of 1997. He knows nothing about spectator sports and likes it that way. (On a good day, he can watch a game being played on TV and tell you what sport it is, usually on the first guess.) Mark currently works as a designer and developer for Object Products Inc. in Fort Collins, CO. He is also the Webmaster for several organizations, including Project New Miramesi, a Honduras reconstruction project. Reach Mark at mark.johnson@javaworld.com.

URL: http://www.javaworld.com/javaworld/jw-02-1999/jw-02-beans.html
Last modified: Tuesday, January 26, 1999