# XML - Application Programming Interfaces (APIs)

*Prepared by Jeff Hunter, Sr. DBA* *12-OCT-2002*

## Overview

The most important decision you'll make at the start of an XML project is the application-programming interface (API) you'll use. Many APIs are implemented by multiple vendors, so if the specific parser gives you trouble you can swap in an alternative, often without even recompiling your code. However, if you choose the wrong API, changing to a different one may well involve redesigning and rebuilding the entire application from scratch. Of course, as Fred Brooks taught us, “In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.… Hence plan to throw one away; you will, anyhow. ” [1] Still, it is much easier to change parsers than APIs.

There are two major standard APIs for processing XML documents with Java, the Simple API for XML (SAX) and the Document Object Model (DOM), each of which comes in several versions. In addition there are a host of other, somewhat idiosyncratic APIs including JDOM, dom4j, ElectricXML, and XMLPULL. Finally each specific parser generally has a native API that it exposes below the level of the standard APIs. For instance, the Xerces parser has the Xerces Native Interface (XNI). However, picking such an API limits your choice of parser, and indeed may even tie you to one particular version of the parser since parser vendors tend not to worry a great deal about maintaining native compatibility between releases. Each of these APIs has its own strengths and weaknesses.

## SAX

SAX, the Simple API for XML, is the gold standard of XML APIs. It is the most complete and correct by far. Given a fully validating parser that supports all its optional features, there is very little you can’t do with it. It has one or two holes, but they're really off in the weeds of the XML specifications, and you have to look pretty hard to find them. SAX is a event driven API. The SAX classes and interfaces model the parser, the stream from which the document is read, and the client application receiving data from the parser. However, no class models the XML document itself. Instead the parser feeds content to the client application through a callback interface, much like the ones used in Swing and the AWT. This makes SAX very fast and very memory efficient (since it doesn’t have to store the entire document in memory). However, SAX programs can be harder to design and code because you normally need to develop your own data structures to hold the content from the document.

SAX works best when your processing is fairly local; that is, when all the information you need to use is close together in the document. For example, you might process one element at a time. Applications that require access to the entire document at once in order to take useful action would be better served by one of the tree-based APIs like DOM or JDOM. Finally, because SAX is so efficient, it’s the only real choice for truly huge XML documents. Of course, “truly huge” has to be defined relative to available memory. However, if the documents you're processing are in the gigabyte range, you really have no choice but to use SAX.

## DOM

DOM, the Document Object Model, is a fairly complex API that models an XML document as a tree. Unlike SAX, DOM is a read-write API. It can both parse existing XML documents and create new ones. Each XML document is represented as Document object. Documents are searched, queried, and updated by invoking methods on this Document object and the objects it contains. This makes DOM much more convenient when random access to widely separated parts of the original document is

required. However, it is quite memory intensive compared to SAX, and not nearly as well suited to streaming applications.

## JAXP

JAXP, the Java API for XML Processing, bundles SAX and DOM together along with some factory classes and the TrAX XSLT API. (TrAX is not a general purpose XML API like SAX and DOM. I'll get to it in Chapter 17.) It is a standard part of Java 1.4 and later. However, it is not really a different API. When starting a new program, you ask yourself whether you should choose SAX or DOM. You don't ask yourself whether you should use SAX or JAXP, or DOM or JAXP. SAX and DOM are part of JAXP.

## JDOM

JDOM is a Java-native tree-based API that attempts to remove a lot of DOM's ugliness. The JDOM mission statement is, "There is no compelling reason for a Java API to manipulate XML to be complex, tricky, unintuitive, or a pain in the neck," and for the most part JDOM delivers. Like DOM, JDOM reads the entire document into memory before it begins to work on it; and the broad outline of JDOM programs tends to be the same as for DOM programs. However, the low-level code is a lot less tricky and ugly than the DOM equivalent. JDOM uses concrete classes and constructors rather than interfaces and factory methods. It uses standard Java coding conventions, methods, and classes throughout. JDOM programs often flow a lot more naturally than the equivalent DOM program.

I think JDOM often does make the easy problems easier; but in my experience JDOM also makes the hard problems harder. Its design shows a very solid understanding of Java, but the XML side of the equation feels much rougher. It's missing some crucial pieces like a common node interface or superclass for navigation. JDOM works well (and much better than DOM) on fairly simple documents with no recursion, limited mixed content, and a well-known vocabulary. It begins to show some weakness when asked to process arbitrary XML. When I need to write programs that operate on any XML document, I tend to find DOM simpler despite its ugliness.

## dom4j

dom4j was forked from the JDOM project fairly early on. Like JDOM, it is a Java-native, tree-based, read-write API for processing generic XML. However, it uses interfaces and factory methods rather than concrete classes and constructors. This gives you the ability to plug in your own node classes that put XML veneers on other forms of data such as objects or database records. (In theory, you could do this with DOM interfaces too; but in practice most DOM implementations are too tightly coupled to interoperate with each other's classes.) It does have a generic node type that can be used for navigation.

## ElectricXML

ElectricXML is yet another tree-based API for processing XML documents with Java. It's quite small, which makes it suitable for use in applets and other storage limited environments. It's the only API I mention here that isn't open source, and the only one that requires its own parser rather than being able to plug into multiple different parsers. It's gained a reputation as a particularly easy-to-use API. However, I'm afraid its perceived ease-of-use often stems from catering to developers' misconceptions about XML. It is far and away the least correct of the tree-based APIs. For instance, it tends to throw away a lot of white space it shouldn't; and its namespace handling is poorly designed. Ideally, an XML API should be as simple as it can be and no simpler. In particular, it should not be simpler than XML itself is. ElectricXML pretends that XML is less complex than it really is, which may work for a while as long as your needs are simple, but will ultimately fail when you encounter more complex documents. The only reason I mention it here is because the flaws in its design aren't always apparent to casual users; and I tend to get a lot of e-mail from ElectricXML users asking me why I'm ignoring it.

## XMLPULL

SAX is fast and very efficient, but its callback nature is uncomfortable for some programmers. Recently some effort has gone into developing pull parsers that can read streaming content like SAX does, but only when the client application requests it. The recently published standard API for such parsers is XMLPULL. XMLPULL shows promise for the future (especially for developers who need to read large documents quickly but just don't like callbacks). However, pull parsing is still clearly in its infancy. On the XML side, namespace support is turned off by default. Even worse, XMLPULL ignores the DOCTYPE declaration, even the internal DTD subset, unless you specifically ask it to read it. From the Java side of things, XMLPULL does not take advantage of polymorphism, relying instead on such un-OOP constructs as int type codes to distinguish nodes instead of making them instances of different classes or interfaces. I don't think XMLPULL is ready for prime time quite yet. However, none of this is unusual for such a new technology. Some of the flaws I cite were also present in earlier versions of SAX, DOM, and JDOM and were only corrected in later releases. In the next couple of years, as pull parsing evolves, XMLPULL may become a much more serious competitor to SAX.

## Data Binding

Recently, there's been a flood of so-called data binding APIs that try to map XML documents into Java classes. While DOM, JDOM, and dom4j all map XML documents into Java classes, these data binding APIs attempt to go further, mapping a Book document into a Book object rather than just a generic Document object, for example. These are sometimes useful in very limited and predictable domains. However, they tend to make too many assumptions that simply aren't true in the general case to make them broadly suitable for XML processing. In particular, these products tend to implicitly depend on one or more of the following common fallacies:

- Documents have schemas or DTDs.
- Documents that do have schemas and/or DTDs are valid.
- Structures are fairly flat and definitely not recursive; that is, they look pretty much like tables.
- Narrative documents aren't worth considering.
- Mixed content doesn't exist.
- Choices don't exist; that is, elements with the same name tend to have the same children.
- Order doesn't matter.

The fundamental flaw in these schemes is an insistence on seeing the world through object-colored glasses. XML documents can be used for object serialization, and in that use-case all these assumptions are reasonably accurate; but XML is a lot more general than that. The large majority of XML documents cannot be plausibly understood as serialized objects, though a lot of programmers approach it from that point of view because that's what they're familiar with. When you're an expert with a hammer, it's not surprising that world looks like it's full of nails.

The fact is, XML documents are not objects and schemas are not classes. The constraints and structures that apply to objects simply do not apply to XML elements and vice versa. Unlike Java objects, XML elements routinely violate their declared types, if indeed they even have a type in the first place. Even valid XML elements often have different content in different locations. Mixed content is quite common. Recursive content isn't quite as common, but it does exist. A little more subtly, though even more importantly, XML structures are based on hierarchy and position rather than the explicit pointers of object systems. It is possible to map one to the other, but the resulting structures are ugly and fragile; and you tend to find that when you're finished what you've accomplished is merely reinventing DOM. XML needs to be approached and understood on its own terms, not Java's. Data binding APIs are just a little too limited to interest me, and I do not plan to treat them in this book.