

# Notes on XML

March 5, 2003

## 1 Intro

XML is a language for describing data. In fact, it's not really a language, but more like a meta-language. That is, a set of rules for describing how to build a language. Even though it looks a lot like HTML, they are quite different. In HTML, a tag like `<b> hello </b>` has a clear meaning: make 'hello' boldface. In XML, a tag like `<author>Tolkien</author>` does not have a set interpretation (or *semantics*); instead, the meaning of the author tag is defined by the XML document itself. This can seem confusing, but some examples will help make it clear.

```
<?xml version="1.0"?>
<book>
  <author> J.R.R. Tolkien </author>
  <title> The Lord of the Rings </title>
  <volumes>
    <volume> Fellowship of The Ring </volume>
    <volume> The Two Towers </volume>
    <volume> Return of the King </volume>
  </volumes>
  <price> 14.95 </price>
  <publisher> Ballantine </publisher>
  <isbn> 0345340426 </isbn>
</book>
```

This simple XML document describes the book the Lord of the Rings. The document has two components: *tags* and *content*. The content is the data in our document; the author, title, and so on. The tags are *metadata*. That is, data about data. They serve to help an application make sense of the data. (In this limited sense, HTML tags also do this; the tags in `<b> hello </b>` just indicate that 'hello' should receive a particular sort of treatment.)

XML requires that every opening tag (e.g. `<volume>`) have a corresponding closing tag (`</volume>`). Everything between an opening and closing tag (including the content and the tags) is called an *element*. For example, `<volume> Return of The King </volume>` is an element, as is everything between `<volumes>` and `</volumes>` inclusive, as is everything between `<book>` and `</book>`.

Notice that elements must be nested: for example,

```
<element1>
  <element2>
    content
  </element2>
</element1>
```

is legal XML, whereas

```
<element1>
  <element2>
    content
  </element1>
</element2>
```

is not. It can help to think of elements as a scope, much like the scoping operators in a programming language. In this case, element2 begins within the scope of element1, and so element2 has to end before element1 ends.

Tags must begin with a letter, and can contain all the same characters as Java variables (letters, numbers, -, underscore, etc. No spaces.).

This regular structure allows us to construct a tree representing an XML document; this will prove to be very useful in parsing and analyzing XML. For example, our Tolkien doc looks like:

```

      |-- author -> tolkien
      |
      |           |-- volume1 -> fellowship of the Ring
      |-- volumes -| volume2 -> The Two Towers
book -- |           |-- volume3 -> The Return of the King
      |
      |--Publisher -> Ballantine
      |
      |-- isbn -> 0345340426
```

The top element (book) is referred to as the *root element* or the *document element*. The tags are referred to as *nodes* (just like normal tree terminology) and the content is stored at the *leaves*.

## 2 More on elements

Sometimes an element has no content - just a tag. For example:

```
<book>
  <author> Tolkien </author>
```

```
<co-author> </co-author>
...
</book>
```

This is called an *empty element*. There's a shorthand for this: we can write `<co-author />` instead. (the space before `'/'` is not essential, but makes it easier to read.)

Element tags can also contain attributes and values; this is useful if you want to specify that an object belongs to one of a few types. For example,

```
<book genre="fantasy" size="large"> ...
</book>
```

Note that the values are enclosed in double quotes (you can also use single quotes, in case your value has a `"` character in it).

The decision whether to use attributes and values or sub-elements is largely a stylistic one; typically, if an variable can only take on one of a few values, an attribute makes sense. If the number of genres is large (or will be extensible), a sub-element makes more sense. Also, order is preserved for sub-elements, whereas attributes and values get treated like a hash table. In general, parameters of an element should be treated as attributes, whereas features of an element should be treated as sub-elements. Most of the rules you've learned for object-oriented design apply here.

```
<book>
  <genre>fantasy </genre>
  ...
</book>
```

Is also valid.

A particularly useful attribute is the ID tag - this lets you assign a unique identifier to an element, and then refer back to it later in the XML document. for example,

```
<volume id="book1"> Fellowship of the Ring </volume>
<volume id="book2"> The Two Towers. Read this book after you've
finished <volumeref idref="book1 />.</volume>
```

Here we used the `*ref` tag to refer to a previous volume; we provide the XML parser with the information that this is a reference to a volume and give it the unique id. Notice that this is an empty element, so we close it with `/i`.

### 3 Document Prolog

An XML document begins with a header known as a *document prolog*, which identifies that the document is in XML and points to definitions of the allowable tags.

Here's an example of a document prolog:

```

<?xml version="1.0" encoding="US-ASCII" standalone="no">
<!DOCTYPE book
  PUBLIC "-//USF //DTD Book 1.8//EN"
  "http://www.foobar.com/DTDs/lotr.dtd"
 [
  <!ENTITY jrirt "J.R.R. Tolkien">
  <!ENTITY elvish-key "elvish.xml">
 ]>

```

We can decompose this into parts:

The first line is the XML declaration. It indicates that the document is XML, the version (1.0 is the only version to date), the encoding (ASCII is default; if you use a non-English character set, you would indicate it here.) and whether the document is standalone, or whether an extra document (usually a DTD - more on that in a bit) is needed.

The second line is called the *document type declaration*; it just says what the type of the root element is; in this case book. This lets the XML processor know what sort of XML document it will be dealing with.

Following this are two lines that designate a *document type definition* (DTD). A DTD describes how a particular XML document is laid out; for example, what elements a book can have, what elements are sub-elements of other elements, what values an attribute can have, and so on. It's like a grammar.

The first line indicates the type of DTD to use and its encoding. The second is a URL that tells the XML processor where to find this DTD. (More on DTDs in a bit.)

The final bit is what's called the *internal subset*; these are basically additional definitions on top of those specified in the DTD. They're called entities. They're very similar to macros in C - we'll talk more about them below.

All of the lines in the prolog except for the first are optional - the bare minimum that a file must have is `<?xml version="1.0">`

## 4 Namespaces

Sometimes you'll want to use more than one DTD within an XML document. for example, say you have a DTD describing books and a DTD describing people. In the `<author>` element, you might want to refer to the 'people' definitions (much as you would use different packages in Perl or Java). This is done by prefixing a tag with the namespace, followed by a colon. For example,

```

<book xmlns:people = "http://www.foobar.com/names">
<title> Lord of the Rings </title>
<author> <people:name> J.R.R.Tolkien </people:name>
<people:title> Sir </people:title>
...
</book>

```

In order to use a namespace, you have to declare it first. We did that above with the `xmlns` attribute inside the `book` tag. It declares the 'people' namespace and points to the appropriate DTD. The definition doesn't have to be at the root document; you might find it useful to introduce a separate namespace for a particular subsection of a document. Following the standard XML scoping rules, that namespace will remain usable within the element where it is declared.

If we only want one default namespace, we can leave out the post-colon identifier; the XML processor will then assume that everything is defined within that namespace. For example:

```
<book xmlns="http://www.foobar.com/names"> ... </book>
```

## 5 Entities

We saw a couple of entities defined in the prolog. For example, we had:

```
<!ENTITY jrirt "J.R.R. Tolkien">
<!ENTITY elvish-key "elvish.xml">
```

We can then use these within a later element, adding an `&` to the beginning and a semicolon to the end. For example:

```
<book> ...
```

```
<description> the Author of The Lord of the Rings is &jrirt; he
invented a grammar and semantics for Elvish, which can be found at
&elvish-key;
</description>
```

The XML parser will replace entities with the objects they refer to.

There are also entities to represent those characters that XML wants to treat as special, such as `<`, `>`, `"`, `'`, and `&`. They are `&lt;`, `&gt;`, `&quot;`, `&apos;`, and `&amp;`, respectively. Also, for non-ASCII characters, you can give the Unicode number corresponding to a character, preceded by `&#`. for example, `&#241` is an `n` with a tilde over it.

If you have large amounts of character data that you want to include (for example, you have HTML formatting instructions that you don't want treated as XML), you can use `CDATA`. For example:

```
<![ CDATA [ <b> this text will be bold <h2> and this text
large</h2> </b> ] ]>
```

Note the two sets of brackets. Everything inside the brackets will be ignored by the parser. This is nice when you need to include code in an XML document.

Declared entities can also contain markup (tags), in addition to text. For example, you might want to do:

```
<!ENTITY special-price "<dollar-amount> 14.95 </dollar-amount>">
<sales-tag> This week, the book is on sale for the low, low price of
&special-price; ! that's right, just &special-price;! can you believe
it? When was the last time you saw this book for only &special-price;?
```

Now you can change both the price itself, and how the price is marked up, all within one tag.

You might also want to make a set of external entities. Let's say you use the same set of entities for all of your data. Rather than putting these tags at the beginning of each XML file (lots to maintain), you can put the entity tags in an external file, and then reference it in the document prolog.

For example, let's say all your documents have a common header and footer. You could do:

```
<?xml version="1.0">
<!DOCTYPE letter SYSTEM "http://www.foobar.com/DTD"
 [
   <!ENTITY header "header.xml">
   <!ENTITY footer "footer.xml">
 ]>
<letter>
  &header;
  Dear Sir ...
  &footer;
</letter>
```

SYSTEM indicates that the entities are defined externally (same with the SYSTEM in the document type declaration - it says that the DTD is to be found elsewhere.)

There's also a more generalized way of referring to external resources, using the PUBLIC keyword. This takes two values, a URI (uniform resource indicator) and a URL. We saw this previously with the book example:

```
<!DOCTYPE book
PUBLIC "-//USF //DTD Book 1.8//EN"
"http://www.foobar.com/DTDs/lotr.dtd"
```

the first string is a Uniform Resource Indicator (URI) - this is meant to be a generalized naming scheme, akin to people's SSNs. The idea is that the URI provides a layer of abstraction; even if an object moves, it will keep the same URI.

The format is: **+,-//Owner ID // Description // Language**

The + or - indicates whether the document is formally registered with a naming service like ISO. An XML processor needs to know how to look this information up in a catalog that maps URIs to locations.

Now, this is more flexible, because the URI doesn't specify a hostname, just a key to look up the DTD. However, this means that whatever is processing this

XML needs to know how to do this. In practice, this is often not implemented, and so browsers will default to using the URL.

## 6 Linking to other documents

As with HTML, you can embed links to other documents within XML. However, XML links are much more expressive than HTML links. For example:

```
<image
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:type="simple"
  xlink:href="logo.gif"
  xlink:show="embed"
/>
```

This specifies that a) we'll use the xlink namespace, b) this is a 'simple' link (there's also complex links, but we'll avoid these) c) a URL for finding the image, and d) how it should be displayed - 'embed' says fetch it immediately and place it in the page. we could also do:

```
<doclink
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:type="simple"
  xlink:href="http://www.cs.usfca.edu"
  xlink:show="replace"
  xlink:actuate="onRequest"
>click here</doclink> to get to the USF CS homepage.
```

In this case, `xlink:show` was set to 'replace', indicating that this document should replace the previous one, and `xlink:actuate` was set to 'onrequest', which says that the link should be followed on the user's request.

Notice that this is still a high-level description; this doesn't say anything about how a browser (for example) should implement this.

## 7 Display

there are several ways in which you can specify how an XML document should be rendered. The simplest, which also works with HTML, are Cascading Style Sheets (CSS) - these specify how a tag should be rendered - for example, that `<b>` should use a particular font size and type.

A more interesting and powerful way to do this is to specify a transformation. This is an XML document that describes how to turn an input XML document into an output XML document with different tags (such as HTML). To do transforms, we need to first talk a bit about Document Models and Schema.

## 8 Document Models and DTDs

A document model lets you specify the rules for your own particular flavor of XML - things like: do books have to have an author element? What are valid values for 'genre'? Can books have more than one title?

This feature is what makes XML so powerful - rather than having a preset, general-purpose set of tags and elements that you have to scrunch into a shape that fits your needs (HTML, anyone?), you can design your own language that fits your problem.

The document model is also essential for programs that have to parse and manipulate XML. Once a parser has the document model, it becomes much easier both to parse the document and to check for errors. Also, it tells an agent how to generate XML so that other programs can read it.

### 8.1 DTDs

OK, we've been talking about DTDs, saying we'll get to it eventually, so what are these things?

A DTD defines the set of allowable elements (the vocabulary), the ways in which elements can be nested (the grammar), and the values attributes can take on (more vocabulary) and their default values.

#### 8.1.1 Entity declarations

We talked about entities before - remember, they're like macros. You can also include them in a DTD.

#### 8.1.2 Element Declarations

An element declaration looks like this:

```
<!ELEMENT book (author | volumes | isbn | title | price)* >
```

This says that there's an element called book, and that it can contain 0 or more author, volumes, isbn, title, and price sub-elements. This third field is called the *content model* (I know, all this jargon is really irritating ...) Notice that the content model uses the standard regular expression syntax - this means that you can specify that an element contains one or more (+), 0 or 1 (?), and use AND (,) and OR (—) to specify combinations.

So we might say:

```
<ELEMENT book (author+ , subtitle?, title, price+, isbn, (volumes | description)*)>
```

This says that a book must have at least one author, exactly one title, at least one price, an optional (0 or 1) subtitle, exactly one isbn, 0 or more volumes, and zero or more descriptions.

<!ELEMENT author ALL> says that author can contain any combination of sub-elements.

<!ELEMENT co-author EMPTY> says that an element must be empty.

<!ELEMENT title (#PCDATA)> says that a title can only contain character strings (no sub-elements).

### 8.1.3 Attribute Declarations

Once you're done describing how elements fit together, you're ready to say what attributes and values go with each element.

The general syntax is:

```
<!ATTLIST element-name
    name1 type2 descrip1
    name2 type2 descrip2
    ...
>
```

The description part specifies whether an attribute is required, and what the default values are. for example:

```
<!ATTLIST book
    genre (fantasy | sci-fi | mystery | horror) 'fantasy'
    id ID #REQUIRED
>
```

Genre can take one of four values; fantasy is the default. id is specified to be of type ID, and is required - this ensures that every book is given a unique identifier.

Data types are: CDATA (character data), NMTOKEN (a string w/o whitespace), ID, IDREF (reference to an ID), ENTITY (refers to a defined ENTITY - see above), or a list of keywords. There are others too - take a look at the XML spec for all the types.

#REQUIRED says that the programmer must sat a value; #IMPLIED says that the attribute is optional and that there is *no* default value.

For example:

```
<!ATTLIST book
    other-media (cd | pdf | audio-tape) #IMPLIED
>
```

We could use this to indicate that the book existed on non-print media.

### 8.1.4 An Example

So here's a possible DTD for our book document:

```

<!-- BTW, here's how to add a comment. This is our book DTD --!>
<!ELEMENT book (author+ , subtitle?, title, price+, publisher+, isbn,
(volumes | description)*)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT volumes volume+>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ATTLIST book
    genre (fantasy | sci-fi | mystery | horror) 'fantasy'
    id ID #REQUIRED
>

```

### 8.1.5 XML Schema

For several reasons, some folks don't like DTDs. The syntax is odd, and not like regular XML. It can be hard to specify complex patterns, and they can be unwieldy as they get large.

So XML Schema have been proposed as a way of dealing with these problems. XML Schema provide a way for you to describe your data using XML directly. XML Schema are a relatively new thing, and the standard is still evolving. I'm not going to go into more detail on Schema because of that. Plus, we've got to get to ...

## 9 Transforming XML

One of the coolest things about XML and its regular structure is the fact that you can transform it from one representation to another automatically. To do this, you create a simple XML-based (of course) document in a language called XSLT (Extensible Style Language Transformations).

This means that if you get your data all nicely packaged into one XML format and then FooBar.com comes out with the new hot XML format next week, you don't have to start over, or write a bunch of hand-rolled programs to get your data in the right format. Instead, you just need to make an XSLT document, then pass the two to a transformation engine. You can also use XSLT with servers that dynamically generate XML to get the data back in a form that's useful for your needs. In other words, a server can just plan to send everyone back XML. A client who wants the data in another format can just pass the server an XSLT file and the server will transform the data for the client. No need to write your own parser, or have a big heavyweight plug-in on the client side reformatting your data. If you change the way your data should be presented, just change the XSLT file.

In order to understand XSL transforms, we first need to remember that when XML is parsed, we get a tree. This tree has elements as nodes and text at the

leaves. Attributes are also considered to be at the leaves.

As a reminder, let's look at a simple tree for this XML document:

```
<?xml version='1.0'>
<book genre='scifi'>
  <author> William Gibson </author>
  <title> Neuromancer </title>
  <price> 9.95 </price>
</book>
```

|--> genre='scifi'

|

|-- author --> William Gibson

root--- book ---|

|-- Title --> Neuromancer

|

|--Price --> 9.95

One interesting wrinkle: above the 'book' node is an abstract node referred to as the *root node*. It encompasses everything about the document, including comments made outside of the scope of the book element.

The trick to XSL transforms is to write a rule for each of the different levels in the tree, plus a default rule for levels that don't need special attention.

These rules will have three components. First, a matching element that will specify which nodes should have this rule applied to them. Second, a set of operators to apply to generate a result. Third, instructions on how to deal with the children of a node.

Not surprisingly, the rules will be expressed as an XML document. It will start out with:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform/"
  version="1.0">
```

The first line should look familiar by now. The second line is an element that specifies the namespace for the XSL transform language. (this is usually a good URL to use.) The transform will happen within the scope of this element.

Now for a matching rule. Let's start with 'author';

```
<xsl:template match="author">
  <i>
    <xsl:apply-templates />
  </i>
</xsl:template>
```

This will italicize the author and apply any templates to the children of the author node. Notice that all we're doing here is asking the transformer to insert some HTML at this point.

We can also write a rule to match the root document. This will be useful for setting up the beginning of the document, and finishing things off at the end.

```
<xsl:template match='''>
<html>
<body>
  <xsl:apply-templates />
</body>
</html>
</xsl:template>
```

This will begin by processing the root node, printing out the HTML setup, and then process the children of root (that's what `<xsl:apply-templates />` does). A complete stylesheet would also have similar rules for title, book, genre, and price.

This works fine, but so far it's only useful for very simple documents. Luckily, XSLT is much more powerful.

First off, you can match on a wide variety of different features of a node, including attribute type, parent, child, multiple elements, and many more. Here's a few examples:

```
<xsl:template match="title | author">
  <b>
    <xsl:apply-templates/>
  </b>
</xsl:template>
```

matches title or author.

```
<xsl:template match="volumes/volume">
  <h3>
    <xsl:apply-templates/>
  </h3>
</xsl:template>
```

This matches a volume node whose parent is a volumes node.

```
<xsl:template match="*/*/title">
  <h2>
    <xsl:apply-templates/>
  </h2>
</xsl:template>
```

this will match a title node that is one level away from the root node - i.e. it is the child of a book node, which is the child only of the root node.

```

<xsl:template match="text()">
  <font>
    <xsl:apply-templates/>
  </font>
</xsl:template>

```

This will match the text nodes at the leaves.

```

<xsl:template match="book[genre='fantasy']">
  This is a fantasy book! <p> <center>
    <xsl:apply-templates/>
  </center>
</xsl:template>

```

This matches book elements whose genre attribute is fantasy. There are many other matching rules - the resources on the class website list them.

If more than one rule can apply to a node, the most specific is used. This means that hierarchical rules outweigh rules that match a node, direct matches outweigh wildcards, and rules that match both element and attribute outweigh element matches.

## 9.1 Control structures

You can also embed control structures within a match. Rather than writing a complex expression, you can write the equivalent of if and switch.

For example,

```

<xsl:template match="*">
  <xsl:if test="self::[id < '4']">
    ID is less than 4
  <xsl:apply-templates/>
</xsl:if>
</xsl:template>

```

Self refers to the node being processed. Here we're testing to see if the ID of the node is less than 4.

```

<xsl:template match="*">
  <xsl:choose>
    <xsl:when test="self::title">
      Title node
    </xsl:when>
    <xsl:when test="self::author">
      Author node
    </xsl:when>
    <xsl:otherwise>
      Some other node.
    </xsl:otherwise>
  </xsl:choose>

```

```

    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

```

And here we have a 'switch' statement. Like switch or case in most languages, once one condition matches, the template 'short-circuits' to the end. Only one match is made.

So far, we've just used `<xsl:apply-templates />` to process subnodes, and allowed the XSLT processor to pass text through unchanged. This works great when our content is text and we just want to change the wrappers around it, but sometimes you need to refer to or manipulate the value at the current node.

The XSL construct for retrieving the value of a node is `<xsl:value-of select=''. '' />` This says to select the value of the currently processed node. Since there's a path in the value of the select attribute, we could also fetch (for example) the parent value, or the root value.

One thing we might want to do is count the number of elements of a given type. XSL provides a `count()` function that lets us do that. For example:

```

<xsl:template match="volumes">
  There are <xsl:value-of select="count(volume)" /> volumes in this
  book.
</xsl:template>

```

or we could compute a discount price on the fly.

```

<xsl:template match="price">
  The discount price is <xsl:value-of select=". - 1.0" />. What a deal!
</xsl:template>

```

This will subtract \$1 from the value of price.

At times, you will want to use iteration within a template. For example, maybe after we print the title of the book, we want to generate a numbered list of the volumes. To do that, we can use `<xsl:for-each>` like so:

```

<xsl:template match=''book''>
  <xsl:for-each select=''volumes/volume''>
    <xsl:value-of select=''position()''>
    <xsl:text /> <br>
  </xsl:for-each>
</xsl:template>

```

There's several new things in here. The first is `xsl:for-each`, which says that when the translator matches a book element, it should apply the following transformation to each subnode of type `volumes/volume`. The second new thing is the use of the `position()` function. This is basically a counter that gives an index to each matched node in the `volumes/volume` list. The third new thing is the `<xsl:text>` tag (note that it's empty) which instructs the matched node to spit out its contents as text.

You can also define temporary variables with in an XSL file. This is useful, for example, if you want to keep a running total, or for any of the many other things that variables are nice for. Here are a few examples:

```
<xsl:variable name='x' /> <!-- defines a variable x --!>
<xsl:variable name='destination' select='acapulco' />
<!-- defines a variable destination, set to the value acapulco --!>

<xsl:variable name =''cond-variable''>
  <xsl:choose>
    <xsl:when test=''$x > 7''>
      <xsl:text>9.99</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>11.99</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>
<!-- this is equivalent to the statement if (x > 7) cond-variable =
  9.99, else cond-variable = 11.99 --!>
```

Note also that you can refer to previously created variables by putting a “\$” in front of them. As you might expect by this point, the scope of a variable is confined to the element in which it is defined.

## 10 Conclusions

This document is designed to be a primer, or an introduction, to the ideas behind XML and XSLT. XSLT in particular is much more complex; this has just hit some of the high notes. This is what I want you to get for this class; enough to know how to create and process fairly simple XML documents. If you want to go deeper (and I recommend it - it’s really interesting stuff!) there’s lots of great resources available. Here are a few:

- O’Reilly’s books. In particular, I’d recommend “Java and XML”, “Learning XML”, and “XSLT”. All their books are great; they make a real effort to tone down the jargon and present things in a straightforward manner, with lots of examples.
- <http://www.zvon.org/xxl/XSLTreference/Output/index.html> - An XSLT reference.
- <http://www.vbxml.com/xsl/tutorials/intro/default.asp> - An XML tutorial; kind of MS/VisualBasic oriented, but good info.
- <http://hotwired.lycos.com/webmonkey/> - lots of web design articles, includes howtos for XML and XSLT.

There's many more available - just type "XSLT tutorial" into Google.