XML Tutorials for Programmers

Ralf I. Pfeiffer
IBM XML Technology Group

Tutorial 2: Writing XML Documents

What does XML do for data?

Here's a simple yet telling example. Old friends and contacts have often asked me for my e-mail address, which they'd lost by changing e-mail software or companies or ISPs. As a programmer I had little sympathy. Why else did we build our collection of awk and perl conversion scripts, as well as back-up, text-delimited copies of important data? Then I had the misfortune of having my disk crash, my company change, and my mail program change. Suddenly, finding myself with an out-of-date e-mail list, my sympathies changed as well!

I now had a problem. How could I maintain my address book while moving between computers, jobs, and mail programs? Every mail program uses a different file format for address book information.

Why do we still have this problem so long after the advent of e-mail? After all, e-mail address book information seems so simple.

Let's look at the following simple address book as defined in an XML document. (This is an oversimplification, of course. Street addresses and other pieces of information are missing.)

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

<addressBook>
  <person>
    <name>
      <family>Wallace</family> <given>Bob</given>
    </name>
    <e-mail>bwallace@megacorp.com</e-mail>
  </person>
</addressBook>
```

The beauty of XML is that you can easily add or invent tags, and thus extend this grammar to include a superset of all the data that any address book or address book-managing application would want. You can even make some tags and attributes optional.

Before the Web existed, there was no real need to make sure all the data on all the desktop islands was accessible and understandable. In fact, there were some territorial advantages for applications having proprietary data formats that were obscure and unpublished.

You can easily extrapolate this address book example to another domain, say word processing. How often have you hunted for an MS Word reader because you received a document in *.doc format? That's a lot of work just to read a document.

Now, the momentum is toward *user choice* in applications. For example, at home, users choose their favorite mail application, browser, and word processor. And at work, they push to have the same choices and functionality. Plug-and-play is becoming a reality with hardware, with applications, and now because of XML, with data.

XML is essentially plug-and-play data, or data that defines itself. For many domains, the application that doesn't parse XML will soon be considered proprietary.

If I know HTML, do I know XML?

You know quite a bit. Almost all of the major constructs in XML are exactly the same as in HTML. The key difference, besides the ability to invent new tags, is that XML is more strict about certain constructs. Regrettably, HTML browsers have historically allowed "bad HTML" and have diverged in their handling of bad code.

However, you *can convert* an HTML document into a well-formed XML document, and we'll go through that process here.

The well-formed XML examples are shown in blue. The bad examples are shown in gray. The next example is parsed by most HTML browsers, but is not well-formed XML.

```
<OL>
  <LI>HTML allows <B><I>improper nesting</B></I>.
  <LI>HTML allows start tags, without end tags, like the <BR> tag.
  <LI>HTML allows <FONT COLOR=#9900CC>attribute values</FONT>
  without quotes.
</OL>
```

In an HTML browser, the above snippet probably renders just as intended with no complaint from the browser. But now let's convert it to **well-formed** XML:

```
<OL>
  <LI>XML requires <B><I>proper nesting</I></B>.</LI>
  <LI>XML requires empty tags to be identified with
  a trailing slash, as in <BR/>.</LI>
  <LI>XML requires <FONT COLOR="#9900CC">quoted attribute
  values</FONT>.</LI>
</OL>
```

That's the concept of a well-formed XML document. And next we'll cover the specific requirements for XML documents, which are also detailed in the W3C XML spec.

Some major differences between HTML and XML

Here are the major differences between HTML and XML.

1. Hierarchical element structure
XML documents must have a strictly hierarchical tag structure. That is, start tags must have corresponding end tags. In XML vocabulary, a pair of start and end tags is called an **element**. Any element must be properly nested within another. As illustrated above, the snippet below is **not well-formed**:

```
<LI>HTML allows <B><I>improper nesting</B></I>.
```

But this snippet is **well-formed**:

```
<LI>XML requires <B><I>proper nesting</I></B>.</LI>
```

for two reasons:

- The <B><I> start tags imply that I is nested within B, so I should end before B, as in </I></B>.
- The <LI> start tag needs a corresponding end tag, </LI>.

2. Empty tags
Empty tags are also allowed as **elements** in XML documents. An empty tag is essentially a start and end tag in one, and is identified by a *trailing slash* after the tag name. For example, this HTML is **not well-formed** XML:

```
<LI>HTML allows start tags, without end tags, like
<BR>tags.</LI>
```

But this is **well-formed** XML:

```
<LI>XML requires empty tags to be identified with a

trailing slash, as in <BR/>.</LI>
```

because the
 is an empty tag and includes the required trailing slash,
. In this way, an XML parser knows immediately not to look for an end tag, because an empty tag is a start and end tag together as one. A start tag and end tag with no data within them are also sometimes referred to as an empty tag, but this is not the precise definition. **3. Single root elements**
XML documents allow only **one** root element. This restriction makes it easier to verify that the document is complete.**4. Quoted attribute values**
All **attribute values** must be within single or double quotes.

The following is **not well-formed**. Note the missing quotes around #9900CC.

```
<FONT COLOR=#9900CC>attribute values</FONT>
```

But this is **well-formed**:

```
<FONT COLOR="#9900CC">quoted attribute values</FONT>
```

5. Declared entities

All **entities** must be declared in a DTD. XML **entities** are analogous to constants in other languages. Entities can be expanded during processing, like a macro-preprocessing capability, saving error-prone duplication of common text. We'll cover DTDs later in this tutorial. We won't cover entities any further, though, since we don't use them in our example. Entities are an important topic, so you may want to refer to the specification.

Now that you know about well-formed XML documents, you're almost ready to start writing one. There are just a few other fine points to cover.

Some minor differences between HTML and XML

The final differences that you need to know are as follows.

1. Case sensitivity

XML tags are case-sensitive. For example, this works fine in HTML:

```
<H1>Remember XML is case-sensitive!</h1>
```

But the H1 and h1 are seen as two entirely different tags in XML, so you must use the same case in both tags. This is correct XML:

```
<H1>Remember XML is case-sensitive!</H1>
```

Tip: Use either upper or lower case for tags. Or use a strict convention, like upper-casing only word boundaries, which is a common programming practice.

2. Relevant white space

White space in the data between tags is relevant, because XML is a data format. However, within the markup itself, and also within quoted attribute values, white space is *normalized*, or removed. For example, in XML, these two poems are the same:

```
<poem form="free ">
To XML or not to XML
That is the question.
</poem>
```

```
<poem

form="free">
To XML or not to XML
That is the question.
</poem>
```

But this poem is different:

```
<poem form="free">
To XML or
not to XML
That is the question.
</poem>
```

Notice that whitespace including newlines within the *data* between start and end tags is relevant.

However, leading and trailing whitespace inside quoted attribute values is *normalized*, or removed. For example form="free " is reduced to form="free". The rules of normalization are somewhat complex, so you may want to refer to the specification regarding attribute value normalization for the details.

(**Technical point**: Even though the parser normalizes whitespace within tags, the files when re-parsed will create the same DOM representation, and so data integrity is maintained.)

The specification provides more detail on white-space as well.

3. Character encoding
XML allows you to specify different character set encodings. The encoding must be identified within the <?xml ?> declaration as an encoding="UTF-8" attribute. An XML processor is required to support 'UTF-8' and 'UTF-16'. For example:

```
<?xml version='1.0' encoding='UTF-8' ?>
```

The IBM XML Parser for Java (XML4J) supports the following encodings:

- US-ASCII
- UCS-2
- UCS-4
- UTF-8
- UTF-16
- Shift_JIS
- EUC-JP
- ISO-2022-JP
- Big5
- GB2312
- ISO-8859-1 through ISO-8859-9

4. Special reserved characters
Several characters are part of the syntactic structure of XML and will not be interpreted as themselves if simply placed within an XML document. You must substitute a special character sequence called an **predefined entity** by XML. More about entities later.

Reserved character	Predefined entity to use instead
<	<
&	&
>	>
'	'
"	"

Only the "<" char seems to be automatically interpreted by most HTML browsers as the start of a markup tag, although the HTML specification may be stricter.

The best difference between HTML and XML: extensibility with validity

Probably the best difference between HTML and XML is this: you can extend XML by creating new tags that make sense for your data.

If you do create new tags, you must define, or constrain, them by writing **grammar rules**, which the tags must obey. Also called a **Document Type Declaration** (**DTD**), these grammar rules are defined in the XML specification. They specify:

- Which tags are allowed within certain other tags
- Which tags and attributes are optional

With regard to a DTD, an XML document can do any of the following:

- Refer to a DTD, using a URI.
- Include a DTD inline as part of the XML document.
- Omit a DTD altogether. Without a DTD, an XML document can be checked for well-formedness, but not for validity.

An XML document is **valid** if its content conforms to the rules in its DTD. Validity allows an application to make sure the XML data is complete, is formatted properly, and has appropriate attribute values. It also allows an application to *construct* valid XML that conforms to that DTD, which is a very powerful feature.

For example, the IBM XML Parser for Java (XML4J) can:

- Ensure that end-users cannot create invalid XML data
- Help users build XML documents by showing what is valid at any given point

Currently the IBM Parser is the only one implementing this powerful capability. More on this later in "Tutorial 3: Parsing XML Using Java."

DTDs: a closer look

Again, a DTD is a *grammar* that describes what tags and attributes are valid in an XML document, and in what context they are valid.

Grammars for languages are often described in an EBNF, or Extended Backus-Naur Form. The XML specification itself uses EBNF to define the allowable XML constructs. EBNF uses production rules where the left side represents a construct, and the right side says what that construct can contain.

Whew, that's a lot of abstract language. Let's look at some examples from our Address Book. For instance, in EBNF, we could use the following rule to say that a person must contain a name, and optionally, an e-mail address.

```
person ::= (name e-mail*)
```

The DTD equivalent **element** declaration looks like:

```
<!ELEMENT person (name, e-mail*)>
```

This DTD statement declares an element called person, which must consist of a name and an optional e-mail.

		Type	Element Declaration	Element Content Model	
<	!	ELEMENT	person	(name, e-mail*)	>

Let's try to avoid reading the specification for now. Instead, refer to the following table for the symbols and special names used in DTD **element** rules. The characters A and B represent an element or an expression found in the Element Content Model.

Element definition	What it means
A?	Matches A or nothing; optional A.
A+	Matches one or more occurrences of A.
A*	Matches zero or more occurrences of A.
A | B	Matches A or B but not both.
A , B	Matches A followed by B, in that order.
(A, B)+	Matches one or more occurrences of (A followed by B). Parentheses are a grouping mechanism; the expression inside parentheses is treated as a unit.
#PCDATA	Keyword matches string data in the current character encoding.

That's enough information for the moment. Please feel free to refer to this table as we continue. Next let's build on our Address Book example.

Sample DTD: Our Address Book DTD

So far, the entire DTD for our simple Address Book is:

```
<?xml encoding="UTF-8"?>
<!ELEMENT addressBook (person)+>
<!ELEMENT person (name,e-mail*)>
<!ELEMENT name (family, given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
```

Note that in the DTD, we may also specify the <?xml encoding="UTF-8"?> expression. However, if we do so, we must also include the encoding in the DTD.

This DTD says that our addressBook is composed of one or more people, where each person has a name, and optional e-mail address. The name is composed of a family name, and a given name. And the

content of each of these is UTF-8 string data.

Creating DTDs and writing valid XML: an interactive example

Let's have some fun. If you return to the tutorial on the XML Web site, you can edit the interactive examples that follow in this section. In each case, when you click the **Parse** button, you will invoke the IBM Parser and get a result. Sometimes we have intentionally planted errors, so watch out!

If our address book DTD has a filename of **ab0.dtd**, we may refer to this DTD from an XML document as follows.

```
<?xml version="1.0"?>
<!DOCTYPE addressBook SYSTEM "http://www.networking.ibm.com/xml/ab0.dtd">
<addressBook>
 <person>
  <name>
   <family>Wallace</family> <given>Bob</given>
  </name>
  <email>bwallace@megacorp.com</email>
 </person>
</addressBook>
```

Parse Reset

If we introduce a second person into our address book, our document is no longer **valid**. What error have we made?

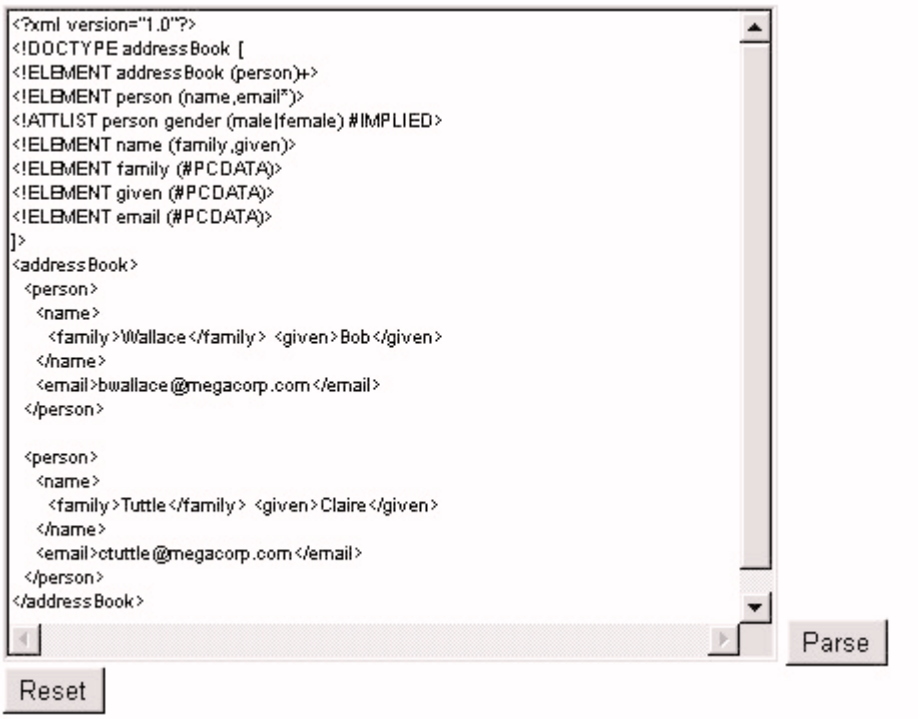
```
<?xml version="1.0"?>
<!DOCTYPE addressBook SYSTEM "http://www.networking.ibm.com/xml/ab0.dtd">
<addressBook>
 <person>
  <name>
   <family>Wallace</family> <given>Bob</given>
  </name>
  <email>bwallace@megacorp.com</email>
 </person>

 <person>
  <name>
   <family>Tuttle</family> <given>Claire</given>
  </name>
  <email>ctuttle@megacorp.com</Email>
 </person>
</addressBook>
```

Parse Reset

1. Click **Parse** to have XML4J parse the document against the ab0.dtd DTD and show the error.
2. See if you can fix the error and **Parse** the document again.

Instead of referring to a DTD via a **URI** (filename, url...), you may simply include the DTD inline as part of your XML document. The example below illustrates this.



The XML declaration

In XML documents and in external DTD documents (in fact, in all external documents), the XML declaration is **optional, yet recommended**. If included, the XML declaration is the first thing in an XML document. As shown in some of the previous examples, it looks like this:

		XML	Version	Encoding		
<	?	xml	version='1.0'	encoding='UTF-8'	?	>

If you do include the XML declaration in your XML document, you must follow some simple rules:

- **<?xml** is **required** as the first characters of the document, with no preceding spaces.
- The **version** attribute is **required**.
- In an external document, such as a DTD, the **encoding** attribute is **required**, and the version is optional (will be inherited from the document).

The XML declaration rules are simpler than they appear in the specification. These rules exist so that XML processors can interpret the XML document correctly. The first few characters specify the character encoding of the file (8- or 16-bit characters, ASCII or EBCDIC) well enough so that a processor can read the rest of the first line, including the "encoding ='xxxx'" attribute. The version is specified so that the XML language may evolve gracefully without breaking existing XML documents. Also, because the XML language is designed to be completely international from the start, any external document that is referenced may have a different encoding.

Attributes

Let's say we want to add an attribute, called **gender** to the element person. (Sometimes the name alone doesn't tell you the gender of the person.)

The basic DTD rule for an **attribute declaration** follows a structure similar to that of **element**.

		Type	Element	Attribute declaration	Attribute definition	
<	!	ATTLIST	person	gender	(male|female) #IMPLIED	>

The (male|female) expression is an **enumerated** type, and #IMPLIED is a keyword designating that this attribute is optional.

In the following *interactive* example, the DTD rule is included for the gender attribute. Can you add the **gender** attribute to the XML part for both people?

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE addressBook [
<!ELEMENT addressBook (person)+>
<!ELEMENT person (name,email*)>
<!ATTLIST person gender (male|female) #IMPLIED>
<!ELEMENT name (family,given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email (#PCDATA)>
]>
<addressBook>
  <person>
    <name>
      <family>Wallace</family> <given>Bob</given>
    </name>
    <email>bwallace@megacorp.com</email>
  </person>

  <person>
    <name>
      <family>Tuttle</family> <given>Claire</given>
    </name>
    <email>ctuttle@megacorp.com</email>
  </person>
</addressBook>
```

Parse

Reset

Default and required attribute values

Let's say we want the gender attribute to default to "unknown". We could express this by adding unknown to the enumerated list, and appending the "unknown" literal in the DTD rule like so:

		Type	Element	Attribute declaration	Attribute definition	
<	!	ATTLIST	person	gender	(male|female|unknown) "unknown"	>

Now, the gender attribute is still optional in the XML part, but if unspecified, it will default to "unknown".

Alternatively, if we want to require the existence of the gender attribute, we use the keyword **#REQUIRED**, like so:

		Type	Element	Attribute declaration	Attribute definition	
<	!	ATTLIST	person	gender	(male|female|unknown) #REQUIRED	>

Now, the gender attribute *must be present* with one of the valid values, within every person tag.

And, finally, what if we wanted to say that if an attribute is present, it is always assigned a particular fixed value? That's possible, too, by using the keyword **#FIXED**.

To understand the finer points of attribute defaults, you may want to refer to the specification.

Attribute types

In our previous examples, we've seen the **enumerated** type of attribute, where the possible attributes are enumerated in a list, delimited with the "|" (or) symbol and surrounded by parentheses. The two other types are the **CDATA** (character data) type and the **tokenized** types.

Use the CDATA type when you want to refer to any character data, potentially even data containing markup. However, certain markup symbols like "<" are not allowed within attributes. So, if you declared an attribute for an element form, say an attribute called method, you could ensure its value was always 'POST', like so:

		Type	Element	Attribute declarationB	Attribute definition	
<	!	ATTLIST	form	method	CDATA #FIXED 'POST'	>

Use the **tokenized** attribute types to represent a fixed set of keyword types with special meanings. Often, we want to uniquely identify instances of a certain element, so it has an attribute with a value that must be unique. In our Address Book example, it would be helpful to be able to uniquely identify and refer to a person, even people with the same name and similar data. This is done using a **tokenized** attribute type called ID.

The **ID** attribute type in the following example, plus the #REQUIRED keyword ensures that every person must have an id attribute whose value is unique within the document.

```
<!ATTLIST person id ID #REQUIRED>
```

Now that we can ensure the uniqueness of element attribute values, we can refer to them. The next tokenized type, called IDREF, does just that.

Each **IDREF** attribute is required to match an ID attribute on some element in the XML document. Similarly, attribute values of type IDREFS must contain whitespace-delimited ID values in the document. So, let's define the ability for a person to link to his or her manager and/or subordinates:

```
<!ELEMENT link EMPTY>
<!ATTLIST link
  manager IDREF #IMPLIED
  subordinates IDREFS #IMPLIED>
```

Notice how we declared an **EMPTY** link element, which can contain attributes that refer to other people.

For details on attribute types, see the specification.

Putting it all together - the complete DTD and XML document

Our complete addressBook DTD, **ab.dtd**, has been defined as follows:

```
<?xml encoding="UTF-8"?>
<!ELEMENT addressBook (person)+>
<!ELEMENT person (name,email*,link?)>
<!ATTLIST person id ID #REQUIRED>
<!ATTLIST person gender (male|female) #IMPLIED>
<!ELEMENT name (#PCDATA|family|given)*>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT link EMPTY>
<!ATTLIST link
  manager IDREF #IMPLIED
  subordinates IDREFS #IMPLIED>
```

Now, your final XML challenge, if you choose to accept it, is to start with the addressBook XML document below, where we left off, and express the fact that Claire is the manager of Bob, and that Bob is a subordinate of Claire. Give it a try below, and hopefully our Parser will help you correct any errors.

```
<?xml version="1.0"?>
<!DOCTYPE addressBook SYSTEM "http://www.networking.ibm.com/xml/ab.dtd">
<addressBook>
 <person gender="male">
  <name>
   <family>Wallace</family> <given>Bob</given>
  </name>
  <email>bwallace@megacorp.com</email>
 </person>

 <person gender="female">
  <name>
   <family>Tuttle</family> <given>Claire</given>
  </name>
  <email>ctuttle@megacorp.com</email>
 </person>
</addressBook>
```

Parse Reset

Does your XML document look something like this? (Note how the link element attributes refer to the contents of an id attribute in the document.)

```
<?xml version="1.0"?>
<!DOCTYPE addressBook SYSTEM "ab.dtd">
<addressBook>
  <person id="B.WALLACE" gender="male">
    <name>
      <family>Wallace</family> <given>Bob</given>
    </name>
    <email>bwallace@megacorp.com</email>
    <link manager="C.TUTTLE"/>
  </person>

  <person id="C.TUTTLE" gender="female">
    <name>
      <family>Tuttle</family> <given>Claire</given>
    </name>
    <email>ctuttle@megacorp.com</email>
    <link subordinates="B.WALLACE"/>
  </person>
</addressBook>
```

Summary

In this tutorial, we have compared and contrasted XML with another markup language that most of us are familiar with, HTML. As you have seen, due to their common heritage (both are simplified subsets of SGML), someone with a cursory knowlege of HTML syntax is instantly conversant in XML syntax. Indeed in one sense there is actually less syntax to learn, since when writing XML you make up your own tags, or use a tagset defined by a DTD in your problem domain.

The DTD (grammar) syntax takes a little getting used to for someone coming from HTML. The good news is that in the near future, you will be able to express the grammar of an XML document by writing the grammar itself in XML in another document. These XML grammars, called schemas, already exist as W3C proposals, but they have not settled completely and are beyond the scope of this tutorial.

Remember that the key differences between HTML and XML are:

- In XML, your tags must be properly nested so they are strictly hierarchical; one must be completely inside another.
- In XML, standalone tags, called empty tags, must have a trailing slash before the closing angle-bracket.
- An XML document must have a single root element, which surrounds all others.
- In XML, attribute values must be quoted.
- XML markup tags are case sensitive.
- Whitespace is relevant between start and end tags.
- XML is extensible and uses a Document Type Declaration (DTD) to define the allowable grammar rules for tags and attributes.

By now it may have occurred to you that since HTML and XML are so similar in structure, and since XML is extensible, that HTML could be represented in XML! Well, by applying the constraints listed above to HTML, you *can* express HTML in XML. In fact, various Document Type Declarations (DTDs) have been written that express the grammar rules of HTML, with varying degrees of "strictness".

In fact, many HTML editors and layout tools are already outputting HTML that observes much of the XML constraints listed above. Now tools, applications, and parsers that understand XML can understand HTML as well. Watch for tools that convert HTML to XML, and XML to HTML.

If you are a programmer interested in creating XML tools, or in XML-enabling your applications, please stay tuned for the next tutorial installment, "Parsing XML Using Java."

What's next?

That's enough XML to make anyone dangerous. Now it's time to take a break before the next task: processing XML documents using Java and the IBM XML Parser for Java.

In the tutorial you have just completed, some important topics have been intentionally omitted. You can read about them in the XML specification as you need them:

- Encoding, Internationalization and Languages
- Entities: Internal and External, the constants and macro-processing of XML
- Processing Instructions - allow documents to contain instructions for applications

Now that you know how to write a valid XML document, the next tutorial will show you how to write a Java program to invoke the IBM XML Parser for Java and manipulate the structure of your address book with the DOM API.

If the next tutorial, "Parsing XML Using Java," isn't available as you read this, please check back very soon.