

# XML

rev. 03/25/2005

Perry Roland  
Digital Library Research & Development Group  
pdr4h@virginia.edu  
982-2702

## WHAT IS XML?

"The Extensible Markup Language (XML) is a subset of SGML ... Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML." -- World Wide Web Consortium (W3C)

## WHY XML?

HTML provides a fixed set of predefined elements --

```
<HTML>
<HEAD>
<TITLE>Home Page</TITLE>
</HEAD>
<BODY>
<H1><IMG SRC="mainlogo.gif"> My Home Page</H1>
<P><EM>Welcome to my Web site!</EM></P>

<H2>Contents</H2>
<P>Please choose one:</P>
<UL>
  <LI><A HREF="writing.html"><B>Writing</B></A>
  <LI><A HREF="family.html"><B>Family</B></A>
  <LI><A HREF="photos.html"><B>Photos</B></A>
</UL>

</BODY>
</HTML>
```

HTML is unsuitable for many documents --

- documents that don't consist of typical components, such as a musical score or set of mathematical equations
- databases, such as an inventory of books
- documents that must be organized in a treelike hierarchical structure, such as a book organized into chapters, etc.

XML provides a syntax, not a limited set of predefined elements. You create your own elements and assign them any names you like - hence the term extensible.

```
<?xml version="1.0"?>
<INVENTORY>
  <BOOK>
    <TITLE>The Adventures of Huckleberry Finn</TITLE>
    <AUTHOR>Mark Twain</AUTHOR>
    <BINDING>mass market paperback</BINDING>
    <PAGES>298</PAGES>
    <PRICE>$5.49</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>Moby-Dick</TITLE>
    <AUTHOR>Herman Melville</AUTHOR>
    <BINDING>trade paperback</BINDING>
    <PAGES>605</PAGES>
    <PRICE>$4.95</PRICE>
  </BOOK>
</INVENTORY>
```

XML is structured hierarchically, with elements nested within other elements and with a single top-level element (known as the document element or root element) that contains all other elements.

## DISPLAYING XML

How can a browser know how to handle and display XML elements? -

- with a style sheet, e.g. Cascading Style Sheet (CSS) or eXtensible Stylesheet Language (XSL)
- data binding
- scripting, e.g. VBScript

## REAL-WORLD USES FOR XML

- representing structured documents
- storing databases
- storing vector graphics: Scalable Vector Graphics (SVG)
- describing multimedia presentations: Synchronized Multimedia Integration Language (SMIL)
- communicating among applications over the web: Simple Object Access Protocol (SOAP)
- Formatting mathematical formulae: Mathematical Markup Language (MathML)
- Describing molecular structures: Chemical Markup Language (CML)
- Encoding genealogical data: Genealogical Data Markup Language (GedML)
- Describing music notation: MusicXML, Music Encoding Initiative (MEI)
- Exchanging real estate transaction information: Real Estate Transaction Standard (RETS)
- Representing theological information: Theological Markup Language (ThML)

## CREATING AND DISPLAYING XML

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

<!-- Filename: Inventory.xml -->
<INVENTORY>
  <BOOK>
    <TITLE>The Adventures of Huckleberry Finn</TITLE>
    <AUTHOR>Mark Twain</AUTHOR>
    <BINDING>mass market paperback</BINDING>
    <PAGES>298</PAGES>
    <PRICE>$5.49</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>Leaves of Grass</TITLE>
```

```

    <AUTHOR>Walt Whitman</AUTHOR>
    <BINDING>hardcover</BINDING>
    <PAGES>462</PAGES>
    <PRICE>$7.75</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>The Legend of Sleepy Hollow</TITLE>
    <AUTHOR>Washington Irving</AUTHOR>
    <BINDING>mass market paperback</BINDING>
    <PAGES>98</PAGES>
    <PRICE>$2.95</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>The Marble Faun</TITLE>
    <AUTHOR>Nathaniel Hawthorne</AUTHOR>
    <BINDING>trade paperback</BINDING>
    <PAGES>473</PAGES>
    <PRICE>$10.95</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>Moby-Dick</TITLE>
    <AUTHOR>Herman Melville</AUTHOR>
    <BINDING>hardcover</BINDING>
    <PAGES>724</PAGES>
    <PRICE>$9.95</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>The Portrait of a Lady</TITLE>
    <AUTHOR>Henry James</AUTHOR>
    <BINDING>mass market paperback</BINDING>
    <PAGES>256</PAGES>
    <PRICE>$4.95</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>The Scarlet Letter</TITLE>
    <AUTHOR>Nathaniel Hawthorne</AUTHOR>
    <BINDING>trade paperback</BINDING>
    <PAGES>253</PAGES>
    <PRICE>$4.25</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>The Turn of the Screw</TITLE>
    <AUTHOR>Henry James</AUTHOR>
    <BINDING>trade paperback</BINDING>
    <PAGES>384</PAGES>
    <PRICE>$3.35</PRICE>
  </BOOK>
</INVENTORY>

```

The document consists of 2 parts

- prolog

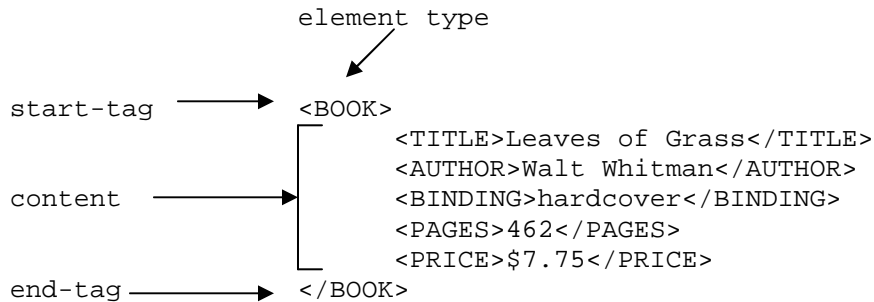
- document element

The prolog may also contain the following optional components

- document type declaration, which defines the type and structure of the document. If used, the document type declaration must follow the XML declaration

- one or more processing instructions, which provide information that the XML processor passes to an application

The document element can contain additional elements which indicate the logical structure of the document and contain the document's information content.



## BASIC XML RULES

- The document must have exactly one top-level element.  
All other elements must be nested within it.
- Elements must be properly nested.  
If an element starts within another element, it must also end within that same element.
- Each element must have both a start-tag and an end-tag  
Unlike HTML, XML doesn't allow you to omit the end-tag -- not even in situations where the browser would be able to figure out where the element ends. Empty elements (those with no content) will be discussed later.
- The element name in the start-tag must EXACTLY match the name in the corresponding end-tag.
- Element names are case-sensitive.  
In fact, all text within XML markup is case-sensitive.

## DISPLAYING XML WITH INTERNET EXPLORER

1. Type example XML above in your favorite text editor. Save as inventory.xml.
2. Open the file in Internet Explorer.
3. Open the file in your text editor and change </TITLE> to </TiTIE>. Save.
4. Reload the file in Internet Explorer.

## DISPLAYING XML WITH INTERNET EXPLORER USING A CASCADING STYLE SHEET

1. Type and save the following CSS as inventory01.css:

```
BOOK
{display:block;
margin-top:12pt;
font-size:10pt}

TITLE
{font-style:italic}

AUTHOR
{font-weight:bold}
```

2. Add following line after the document prolog and before the document element. Save.  
`<?xml-stylesheet type="text/css" href="inventory01.css"?>`
3. Open inventory.xml in IE.
4. Type and save the following CSS as inventory02.css.

```

BOOK
    {display:block;
    margin-top:12pt;
    font-size:10pt}

TITLE
    {display:block;
    font-size:12pt;
    font-weight:bold;
    font-style:italic}

AUTHOR
    {display:block;
    margin-left:15pt;
    font-weight:bold}

BINDING
    {display:block;
    margin-left:15pt}

PAGES
    {display:none}

PRICE
    {display:block;
    margin-left:15pt}

```

5. Edit the stylesheet processing instruction so that it reads:  
`<?xml-stylesheet type="text/css" href="inventory02.css"?>`
6. Open inventory.xml in IE.

## CREATING WELL-FORMED XML DOCUMENTS

An XML document is well-formed when it follows the basic rules for XML given above. The following is NOT a well-formed document. Why?

```

<?xml version="1.0"?>
<BOOK>
  <TITLE>The Adventures of Huckleberry Finn
  <AUTHOR>Mark Twain</TITLE></AUTHOR>
  <BINDING>mass market paperback</BINDING>
  <PAGES>298</PAGES>
  <PRICE>$5.49</PRICE>
</BOOK>
<BOOK>
  <TITLE>Leaves of Grass</TITLE>
  <AUTHOR>Walt Whitman</AUTHOR>
  <BINDING>hardcover</BINDING>

```

```
<PAGES>462</PAGES>
<PRICE>$7.75</PRICE>
</BOOK>
```

## ANATOMY OF AN ELEMENT

- element type or generic identifier (GI)  
name must begin with a letter or underscore ( \_ ), followed by zero or more letters, digits, periods, hyphens, or underscores
- element-type names beginning with “xml” (in any combination of uppercase or lowercase) are “reserved for standardization”.

## ELEMENT CONTENT

- nested elements:

```
<BOOK>
  <TITLE>Leaves of Grass</TITLE>
  <AUTHOR>Walt Whitman</AUTHOR>
  <BINDING>hardcover</BINDING>
  <PAGES>462</PAGES>
  <PRICE>$7.75</PRICE>
</BOOK>
```
- character data:

```
<TITLE>Leaves of Grass</TITLE>
```
- general entity references or character references:

```
<TITLEPAGE>
Author: &author;
Document Name: "How to Enter the &#60; Character"
</TITLEPAGE>
```
- CDATA sections:

```
<TITLEPAGE>
Author: Me
<![CDATA[ Document Name: "How to Enter the < Character" ]]>
</TITLEPAGE>
```
- Processing instructions:

```
<?MyApp Parm1="value1"?>
```
- Comments:

```
<!-- This is a comment. -->
```

## EMPTY ELEMENTS

Some elements are empty, that is, without content. Either place the end-tag immediately after the start-tag, e.g. `<HR></HR>`, or use the special empty-element tag, e.g. `<HR />`. Both have the same meaning. When using the empty-element tag, it is standard practice to include a space between the element name and the /.

## ADDING ELEMENTS

Using inventory.xml:

1. Add comments inside the INVENTORY element.
2. Add the following element at the beginning of each BOOK element:  
`<COVER source="[bookname].gif" />`
3. Add a subtitle element for Moby Dick containing the text "Or, the Whale".

## ATTRIBUTES

In the start-tag of an element, or in an empty-element tag, you can include one or more attribute specifications. An attribute specification is a name-value pair that is associated with the element. The following BOOK element includes two attributes, category and display.

```
<BOOK category="fiction" display="emphasize">
  <TITLE>The Marble Faun</TITLE>
  <AUTHOR>Nathaniel Hawthorne</AUTHOR>
  <BINDING>trade paperback</BINDING>
  <PAGES>473</PAGES>
  <PRICE>$10.95</PRICE>
</BOOK>
```

Attributes provide an alternative way to include information in an element. Typically, the bulk of the data you intend to display is placed in the element's content. Then you use attributes to store various properties of the element not necessarily intended to be displayed. XML, however, makes no distinctions about the type of information that should be stored in attributes or content.

## RULES FOR ATTRIBUTE NAMES

- The name must begin with a letter or an underscore, followed by zero or more letters, digits, periods, hyphens, or underscores.
- Attribute names beginning with 'xml' are reserved.
- A particular attribute name can appear only once in the same start-tag or empty-element tag.

## RULES FOR ATTRIBUTE VALUES

An attribute value is a series of characters delimited with quotes, known as a quoted string or string literal.

- The string can be delimited using either single or double quotes.
- The string cannot contain the same quote character used to delimit it.
- The string can contain character references or references to general internal entities. (More about this later.)
- The string cannot include the < character.
- The string cannot include the & character, except to begin a character or entity reference.

## ADDING ATTRIBUTES

Using inventory.xml:

1. Add a 'born' attribute to each AUTHOR element.

## COMMENTS

A comment begins with `<!--` and ends with `-->`. Between these two delimiters you can place any characters you want -- except the double hyphen, e.g. `--`. Even markup characters forbidden elsewhere, such as `<` and `&` can go here.

Comments may be placed:

- in the document prolog

```
<?xml version="1.0"?>
<!-- Here is a comment in the prolog. -->
<DOCELEMENT>A very simple XML document</DOCELEMENT>
```

- following the document element

```
<?xml version="1.0"?>
<DOCELEMENT>A very simple XML document</DOCELEMENT>
<!-- This comment follows the document element. -->
```

- within an element's content

```
<?xml version="1.0"?>
<DOCELEMENT><!-- This comment is part of the content of the root element. -->
A very simple XML document</DOCELEMENT>
```

Comments may NOT be placed within markup tags, e.g.

```
<?xml version="1.0"?>
<DOCELEMENT <!-- This comment is ILLEGAL! --> >A very simple XML
document</DOCELEMENT>
```

## PROCESSING INSTRUCTIONS

Processing instructions take the general form:

```
<? target instruction ?>
```

where target is the name of the application to which the instruction is directed.

Exactly what happens when a processing instruction is read depends upon the reading application, e.g. the browser. Processing instructions may be inserted anywhere in XML outside other markup -- the same places you can use comments.

## CDATA SECTIONS

One way of getting around the restriction against directly inserting `<` or `&` characters in element content is to use character references (`&#60;` or `&#38;`). If you need to insert many `<` or `&` characters, using references is awkward and makes the data difficult for humans to read. In this case, it's easier to place the text containing the restricted characters inside a CDATA section.

CDATA sections begin with `<![CDATA[` and end with `]]>`:

```

<A-SECTION>
The following is an example of a simple HTML page:
<![CDATA[
<HTML><HEAD><TITLE>My page</TITLE></HEAD>
<BODY><P>Welcome!</P></BODY></HTML>
]]>
</A-SECTION>

```

CDATA sections can be used anywhere that character data can occur, that is, within an element's content, but not within XML markup. They cannot be placed in the prolog or following the document element. The following document contains two illegal CDATA sections:

```

<?xml version="1.0"?>
<![CDATA[ This is ILLEGAL! It's not within element content!]]>
<DOCELEMENT>
    <SUBELEMENT <![CDATA[ ILLEGAL: inside markup!]]> >
    </SUBELEMENT>
</DOCELEMENT>

```

CDATA sections cannot be nested.

## WELL-FORMED-NESS ERRORS

A violation of the well-formed-ness rules is considered a fatal error. When the XML processor encounters a fatal error it must stop normal processing and not attempt to recover.

## CREATING VALID XML DOCUMENTS

Valid XML documents meet a stricter set of criteria than do well-formed documents.

A valid document is a well-formed document that also meets two additional requirements:

- The document prolog must include a proper document type declaration that includes a document type definition (DTD) that defines the structure of the document,
- The rest of the document must conform to the structure defined in the DTD.

## VALIDITY ERRORS

A violation of the validity rules is not a fatal error. An XML processor can simply report the error and attempt to recover from it.

## ADVANTAGES OF MAKING A DOCUMENT VALID

Creating a valid XML document might seem to be a lot of unnecessary trouble: You must first fully define the document's structure in the DTD and then create the document, following all the DTD rules. However, if you want to make sure that your document conforms to a specific structure or standard, the use of a DTD allows an XML processor to guarantee conformance.

Making XML documents valid also ensures uniformity among a group of similar documents. The XML standard defines a DTD as "a grammar for a class of documents".

## ADDING A DTD

The document type declaration goes in the prolog of a valid XML document. It must follow the XML declaration.

```
<?xml version="1.0"?>   DTD can go here
                        |
                        ←
<!-- Filename: Inventory.xml -->   or here
                        |
                        ←
<INVENTORY>
etc.
```

## THE FORM OF THE DOCUMENT TYPE DECLARATION

A document type declaration has the following general form:

```
<!DOCTYPE Name DTD>
```

The *Name* specifies the name of the document element. It must match the document element name exactly.

```
<!DOCTYPE INVENTORY DTD>
```

*DTD* represents the document type definition which defines the document's elements, attributes, etc.

## THE FORM OF THE DOCUMENT TYPE DEFINITION

The DTD consists of a left square bracket, followed by a series of markup declaration, followed by a right square bracket. Markup declarations describe the logical structure of the document. For example,

```
<?xml version="1.0"?>
<!DOCTYPE SIMPLE [
    <!ELEMENT SIMPLE ANY>
]>
<simple>This is an extremely simplistic XML document.</simple>
```

(There's an error in the document above. Can you find it?)

The DTD in this example specifies that the document can contain only elements of type *SIMPLE* and that a *SIMPLE* element can contain *ANY* possible type of content - other defined elements, character data, entity references, CDATA sections, processing instructions, or comments.

A DTD can contain the following types of markup declarations:

- element type declarations which define the types of elements, their order, and contents
- attribute-list declarations which defines the names, the data types, and default values of attributes
- entity declarations which are used to store frequently used blocks of text or to incorporate non-XML data
- notation declarations which describe a data format or identify the program used to process a particular format
- processing instructions
- comments
- parameter entities

## ELEMENT TYPE DECLARATIONS

An element type declaration indicates the name of the element type and the allowable contents of the element (often specifying the order in which child elements can occur). Taken together, the element type declarations map out the entire logical structure of the document.

## THE FORM OF AN ELEMENT TYPE DECLARATION

An element type declaration has the following general form:

```
<!ELEMENT Name contentmodel>
```

*Name* is the name of the element; *contentmodel* is where the element's contents are defined.

```
<?xml version="1.0"?>
<!DOCTYPE COLLECTION [
  <!ELEMENT COLLECTION (CD)+>
  <!ELEMENT CD (#PCDATA)>
  <!-- My audio collection DTD -->
]>

<COLLECTION>
  <CD>Mozart Violin Concertos 1, 2, and 3</CD>
  <CD>Pink Floyd, Dark Side of the Moon</CD>
  <CD>Floyd Cramer, Greatest Hits</CD>
</COLLECTION>
```

## ELEMENT CONTENT SPECIFICATION

You can specify the content of an element -- the *contentmodel* part of the declaration -- in four different ways:

- EMPTY content -- the keyword EMPTY indicates that the element cannot have content.  
<!ELEMENT IMAGE EMPTY>
- ANY content -- the keyword ANY signifies that the element can have any legal type of content. The element can have zero or more child elements, in any order, with or without interspersed character data.  
<!ELEMENT MISC ANY>
- Element content -- the element can contain child elements, but can't directly contain character data.
- Mixed content -- the element can contain any quantity of character data, optionally interspersed with child elements of the specified types.

## SPECIFYING ELEMENT CONTENT

```
<?xml version="1.0">
<!DOCTYPE BOOK [
  <!ELEMENT BOOK (TITLE, AUTHOR)>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT AUTHOR (#PCDATA)>
  <!-- PCDATA stands for "parsed character data" which means
        "this element might contain markup". -->
]>
```

other elements only

parsed character data only

```

<BOOK>
  <TITLE>The Scarlet Letter</TITLE>
  <AUTHOR>Nathaniel Hawthorne</AUTHOR>
</BOOK>

```

A content model can have either of the following basic forms:

- Sequence -- the element must have a specific sequence of child elements.

```

<!DOCTYPE MOUNTAIN [
  <!ELEMENT MOUNTAIN (NAME, HEIGHT, STATE)>
  <!ELEMENT NAME (#PCDATA)>
  <!ELEMENT HEIGHT (#PCDATA)>
  <!ELEMENT STATE (#PCDATA)>
]>

```

- Choice -- the element can have any one of a series of possible child elements

```

<!DOCTYPE FILM [
  <!ELEMENT FILM (STAR|NARRATOR|INSTRUCTOR)>
  <!ELEMENT STAR (#PCDATA)>
  <!ELEMENT NARRATOR (#PCDATA)>
  <!ELEMENT INSTRUCTOR (#PCDATA)>
]>

```

You can modify either of these forms of content model by using the question mark (?), plus (+), and asterisk (\*) characters which have the following meanings:

```

?      Zero or one of the preceding item
+      One or more of the preceding item
*      Zero or more of the preceding item

```

Examples:

```

<!ELEMENT MOUNTAIN (NAME+, HEIGHT?, STATE)>
<!ELEMENT FILM (STAR* | NARRATOR | INSTRUCTOR)>

```

You can also use the ?, +, or \* characters to modify groups of child elements:

```

<!ELEMENT FILM (STAR | NARRATOR | INSTRUCTOR)+>

```

Finally, you can form complex content models by nesting a choice model within a sequence model or a sequence model within a choice model:

```

<!DOCTYPE FILM [
  <!ELEMENT FILM (TITLE, CLASS, (STAR | NARRATOR | INSTRUCTOR))>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT CLASS (#PCDATA)>
  <!ELEMENT STAR (#PCDATA)>
  <!ELEMENT NARRATOR (#PCDATA)>
  <!ELEMENT INSTRUCTOR (#PCDATA)>
]>

```

According to this DTD, both the following documents are legal:

```

<?xml version="1.0"?>
<FILM>
  <TITLE>The Net</TITLE>
  <CLASS>fiction</CLASS>
  <STAR>Sandra Bullock</STAR>
</FILM>

```

```
<?xml version="1.0"?>
<FILM>
  <TITLE>How to Use XML</TITLE>
  <CLASS>instructional</CLASS>
  <INSTRUCTOR>Joe Cool</INSTRUCTOR>
</FILM>
```

## SPECIFYING MIXED CONTENT

If an element has mixed content, it can contain character data. If you specify one or more child element types in the element declaration, it can contain any of those child elements in any order and with any number of repetitions. In other words, with mixed content you can constrain the types of the child elements, but you can't constrain the order or number of occurrences of child element, nor can you make a particular child element type mandatory.

Mixed content models can take either of the following forms:

- Character data only

```
<!ELEMENT SUBTITLE (#PCDATA)>
```

PCDATA stands for “parsed character data” which means “this element might contain markup”.

- Character data plus optional child elements

```
<!ELEMENT TITLE (#PCDATA | SUBTITLE)*>
```

Valid TITLE elements, conforming to this declaration:

```
<TITLE>Moby-Dick <SUBTITLE>Or, the Whale</SUBTITLE></TITLE>
<TITLE><SUBTITLE>Or, the Whale</SUBTITLE> Moby-Dick</TITLE>
<TITLE>Moby-Dick</TITLE>
<TITLE><SUBTITLE>Or, the Whale</SUBTITLE> <SUBTITLE>Something
else</SUBTITLE></TITLE>
<TITLE></TITLE>
```

## DECLARING ATTRIBUTES

An attribute-list declaration has the following general form:

```
<!ATTLIST Name AttDefs>
```

*Name* is the name of the element associated with the attribute(s). *AttDefs* is a series of one or more attribute definitions, one for each attribute.

An attribute definition has the following form:

```
Name AttType DefaultDecl
```

*Name* is the name of the attribute. *AttType* is the attribute type (more later). *DefaultDecl* is the default declaration which indicates whether the attribute is required and provides other information (more later).

Example:

```
<!DOCTYPE FILM [
  <!ELEMENT FILM (TITLE, CLASS, (STAR | NARRATOR | INSTRUCTOR))>
  <!ATTLIST FILM      Class CDATA #IMPLIED
                    Year  CDATA #REQUIRED>
```

```

    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT STAR (#PCDATA)>
    <!ELEMENT NARRATOR (#PCDATA)>
    <!ELEMENT INSTRUCTOR (#PCDATA)>
  ]>

  <FILM Year="1948">
    <TITLE>The Morning After</TITLE>
    <STAR>Morgan Attenbury</STAR>
  </FILM>

```

## ATTRIBUTE TYPE

You can specify the attribute type in three different ways:

- String type -- a string type attribute can be assigned any quoted string (also known as a literal) that conforms to the general rules given earlier.
- Tokenized type -- the values you can assign are constrained in some way.
- Enumerated type -- you can assign a value from a specified list of items.

## SPECIFYING A STRING TYPE

```
<!ATTLIST FILM      Class CDATA #IMPLIED
```

## SPECIFYING A TOKENIZED TYPE

Here's a complete list of the keywords you can use to define tokenized type attributes and the constraints they impose on the attribute values:

- ID -- the attribute must have a unique value. The value must begin with a letter or underscore, followed by zero or more letters, digits, periods, hyphens, or underscores. It can also contain a single colon except in the first position. The attribute's default declaration must be either #REQUIRED or #IMPLIED (more about this later).

```

Example:    <!ATTLIST ITEM      StockCode  ID      #REQUIRED>
            <ITEM StockCode="S123" />

```

- IDREF -- the attribute value must match the value of some ID type attribute elsewhere in the document.

```

Example:    <!ATTLIST ITEM      StockCode  ID      #REQUIRED
            GoesWith        IDREF #IMPLIED>
            <ITEM StockCode="S123" GoesWith="X456" />

```

- IDREFS -- a list of ID references separated by white space characters.

```

Example:    <!ATTLIST ITEM      StockCode  ID      #REQUIRED
            GoesWith        IDREFS    #IMPLIED>
            <ITEM StockCode="S123" GoesWith="X456 Z789" />

```

- ENTITY/ENTITIES -- the attribute value must match the name of an unparsed entity declared in the DTD. An unparsed entity refers to an external file, typically one storing non-XML data (more later).

```

Example:    <!ATTLIST IMAGE      Source      ENTITY      #REQUIRED>
            <IMAGE Source="logo" />

```

- NMTOKEN/NMTOKENS -- the value is a name token, that is, a name which consists of one or more letters, digits, periods, hyphens, or underscores. A single colon may be included except in the first position. NMTOKEN also allows a digit in the first position.

Example:       <!ATTLIST BOOK     ISBN   NMTOKEN       #REQUIRED>  
                   <BOOK ISBN="9-99999-999-9">

## SPECIFYING ENUMERATED TYPES

Enumerated type specifications for attribute values can have either of the following two forms:

- A list of options contained within parentheses separated with | characters

```
<!ATTLIST FILM     Class           (fictional|instructional|documentary)
"fictional">
```

- The keyword NOTATION followed by a list of notation names.

```
<!ATTLIST IMAGE    Source        ENTITY        #REQUIRED
                  Type           NOTATION     (gif|jpg|tif)       #REQUIRED>
```

Assigning a value other than one from the enumerated list results in a validity error.

## THE DEFAULT DECLARATION

The default declaration has four possible forms:

- #REQUIRED -- a value for an attribute with this default declaration must be specified.

```
<!ATTLIST FILM     Class CDATA #REQUIRED>
```

- #IMPLIED -- a value may be included or omitted. If omitted, no default value is supplied to the processor.

```
<!ATTLIST FILM     Class CDATA #IMPLIED>
```

- *AttValue* -- *AttValue* is a default attribute value. A value may be included or omitted. If omitted, the processor will use the default supplied here just as if it occurred in the document.

```
<!ATTLIST FILM     Class CDATA "fictional">
```

The following two elements are equivalent:

```
<FILM>The Graduate</FILM>
<FILM Class="fictional">The Graduate</FILM>
```

- #FIXED *AttValue* -- A value may be included or omitted. If omitted, the processor will use the default value supplied here. If included, the value must match the value supplied here. This is useful primarily for making the document clearer for human readers.

```
<!ATTLIST FILM     Class CDATA #FIXED "documentary">
```

With this declaration, the first element is valid while the second is not.

```
<FILM>The Making of XML</FILM> or <FILM Class="documentary">The Making of
XML</FILM><!-- valid -->
<FILM Class="instructional">The Making of XML</FILM><!-- invalid -->
```

## USING AN EXTERNAL DTD SUBSET

All or part of the document's DTD may be placed in a separate file which is then referred to from the document type declaration. A DTD -- or portion of a DTD -- contained in a separate file is known as an external DTD subset.

To use only an external DTD subset, omit the block of markup declarations within the square brackets and instead include the keyword SYSTEM followed by a quoted description of the location of the separate file that contains the DTD.

Instead of:

```
<?xml version="1.0"?>
<!DOCTYPE simple [
    <!ELEMENT simple ANY>
]>
<simple>This is an extremely simplistic XML document.</simple>
```

Use:

```
<?xml version="1.0"?>
<!DOCTYPE simple SYSTEM "simple.dtd" >
<simple>This is an extremely simplistic XML document.</simple>
```

The file location is known as a system literal. It can be delimited using either single or double quotes and it can contain any characters except the quotes used to delimit it. The system literal specifies the uniform resource identifier (URI) of the file containing the external DTD subset. It may be a fully qualified URI, such as:

```
<!DOCTYPE simple SYSTEM "http://bogus.com/dtds/simple.dtd">
```

or a partial URI that specifies a location relative to the location of the XML document containing the URI, such as:

```
<!DOCTYPE simple SYSTEM "simple.dtd">
```

## CONDITIONALLY IGNORING SECTIONS OF AN EXTERNAL DTD SUBSET

The XML processor can ignore/include a portion of an external DTD subset. You might use an IGNORE section to temporarily de-activate a section of the DTD while you're developing it. To turn it back on, replace the keyword IGNORE with INCLUDE.

```
<![IGNORE[
    <!-- under development -->
    <!ATTLIST BOOK    Category    CDATA "fiction">
    <!ELEMENT TITLE   (#PCDATA)>
    <!ELEMENT AUTHOR  (#PCDATA)>
]]>
```

## USING BOTH EXTERNAL AND INTERNAL DTD SUBSETS

The document contains:

```
<?xml version="1.0"?>
<!DOCTYPE BOOK SYSTEM "file:///Book.dtd" [
    <!ATTLIST BOOK    isbn    CDATA #IMPLIED
                    year    CDATA "2005">
    <!ELEMENT TITLE   (#PCDATA)>
]>
<BOOK year="1998">
    <TITLE>The Scarlet Letter</TITLE>
</BOOK>
```

The DTD file contains:

```
<!ELEMENT BOOK ANY>
<!ATTLIST BOOK    ISBN    NMTOKEN    #REQUIRED>
```

Here's how the XML processor combines internal and external DTD subsets:

- In general, the contents of the two subsets are merged to form the complete DTD.
- If, however, an attribute with the same name and element type is declared more than once, the XML processor uses the first declaration and ignores all subsequent ones.
- The internal DTD subset is considered to come before the external subset (even though the external reference appears first in the document type declaration. Any attribute or entity defined in the internal subset takes precedence over one with the same name and element type declared in the external subset. Re-declaration of an element is illegal.

The way the XML processor combines an internal and external DTD subset lets you use a common DTD as an external DTD subset, but then customize the DTD for the current document by including an internal subset. Your internal subset can add elements, attributes, or entities -- and it can change the definitions of attributes or entities.

## CONVERTING A WELL-FORMED DOCUMENT TO A VALID DOCUMENT

1. Open Inventory.xml in IE. Make sure it is well-formed.
2. Open Inventory.xml in a text editor and add the following DTD subset:

```
<!DOCTYPE INVENTORY [  
    <!ELEMENT INVENTORY (BOOK)*>  
    <!ELEMENT BOOK (TITLE, AUTHOR, BINDING, PAGES, PRICE)>  
    <!ATTLIST BOOK InStock (yes|no) #REQUIRED>  
    <!ELEMENT TITLE (#PCDATA | SUBTITLE)*>  
    <!ELEMENT SUBTITLE (#PCDATA)>  
    <!ELEMENT AUTHOR (#PCDATA)>  
    <!ATTLIST AUTHOR Born CDATA #IMPLIED>  
    <!ELEMENT BINDING (#PCDATA)>  
    <!ELEMENT PAGES (#PCDATA)>  
    <!ELEMENT PRICE (#PCDATA)>  
    <!ATTLIST PRICE Units CDATA "USD">  
>
```

3. If it's missing, add the following SUBTITLE element to the TITLE element of Moby-Dick:  
<SUBTITLE>Or, the Whale</SUBTITLE>
4. Add the InStock attribute to each BOOK element, assigning proper values.
5. Add the Born element to one or more AUTHOR elements.
6. Save as Inventory2.xml.
7. Open Inventory2.xml in IE.
8. Open Inventory2.xml in a text editor.
9. Cut the DTD out and paste it into a new document. Save the new document as Inventory.dtd.
10. Edit the document type declaration of Inventory2.xml to point to the newly-created external DTD subset.
11. Save Inventory2.xml.
12. Open Inventory2.xml in IE.

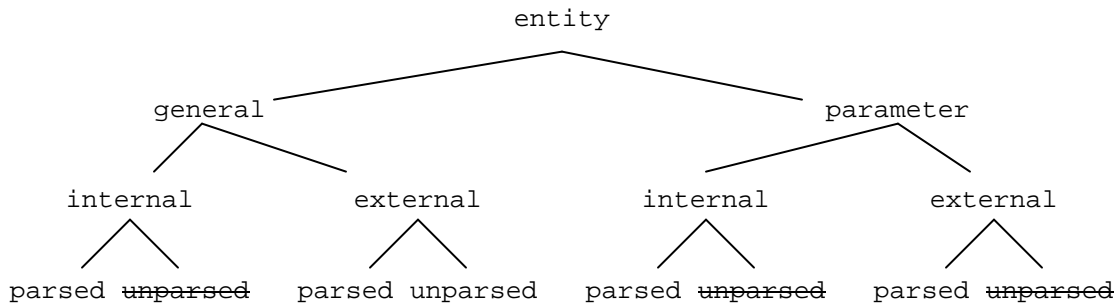
## DEFINING AND USING ENTITIES

Entities can be used as time-savers and as a way to incorporate diverse data types into XML documents. You can define a frequently used block of XML text as an entity, allowing you to quickly insert the text wherever you need it. You can also define an external file as an entity so that you can associate the file's data with your document. Entities are created by declaring them in the DTD.

Entities are classified in three different ways:

- General vs. parameter -- A general entity contains document content, that is, "stuff" (text and non-text) that you can use within the document element. A parameter entity contains XML text that can be inserted within the DTD.
- Internal vs. external -- An internal entity is one contained within the document while an external entity is contained in a separate file.
- Parsed vs. unparsed -- A parsed entity is one that consists of XML text. Its content is parsed by the parser just as if you had typed its contents into the document. An unparsed entity can contain any type of data, but usually non-XML, such as binary data is stored there. Because an unparsed entity typically contains non-XML data, its contents aren't directly inserted into the document. However, you can assign the entity name to an ENTITY or ENTITIES type attribute so that the application can access the entity's name and description and do what it wants with the data.

Theoretically, there are eight potential types of entities:



However, XML does not provide the three entity types that are struck through in the figure, and so XML actually has only five entity types:

- general internal parsed
- general external parsed
- general external unparsed
- parameter internal parsed
- parameter external parsed

## DECLARING A GENERAL INTERNAL PARSED ENTITY

A general internal parsed entity has the following general form:

```
<!ENTITY EntityName EntityValue>
```

- The name follows the same rules as element names, including case-sensitivity.
- The entity can have the same name as a parameter entity, an element, or an attribute.
- The value is a quoted string.
- The string cannot include the ampersand or percent characters except as a character or general entity reference.
- The contents of the string must, of course, be legal for the location where you intend to insert the entity.

```
<!DOCTYPE ARTICLE [
  <!ELEMENT ARTICLE (TITLEPAGE, INTRODUCTION, SECTION*)>
  <!ELEMENT TITLEPAGE (#PCDATA|SUBTITLE)*>
  <!ELEMENT SUBTITLE (#PCDATA)>
  <!ELEMENT INTRODUCTION (#PCDATA)>
  <!ELEMENT SECTION (#PCDATA)>
  <!ENTITY title      "The Story of XML <SUBTITLE>The Future Language of
the Internet</SUBTITLE>">
]>

<TITLEPAGE>
  Title: &title;
  Author: Me
</TITLEPAGE>
```

## DECLARING A GENERAL EXTERNAL PARSED ENTITY

A general external parsed entity has the general form:

```
<!ENTITY EntityName SYSTEM SystemLiteral>
```

```
<!DOCTYPE ARTICLE [
  <!ELEMENT ARTICLE (TITLEPAGE, INTRODUCTION, SECTION*)>
  <!ELEMENT TITLEPAGE (#PCDATA|SUBTITLE)*>
  <!ELEMENT SUBTITLE (#PCDATA)>
  <!ELEMENT INTRODUCTION ANY>
  <!ELEMENT SECTION (#PCDATA)>
  <!ENTITY title      "The Story of XML <SUBTITLE>The Future Language of
the Internet</SUBTITLE>">
  <!ENTITY topics SYSTEM "topics.xml">
]>
```

Here are the contents of the topics.xml file:

```
<HEADING>Topics</HEADING>
The Need for XML
The Official Goals of XML
Real-World Uses for XML
```

Here's part of the document content:

```
<INTRODUCTION>
  Here's what the article covers:
  &topics;
</INTRODUCTION>
```

Is this a valid document?

## DECLARING A GENERAL EXTERNAL UNPARSED ENTITY

A general external unparsed entity has this form:

```
<!ENTITY EntityName SYSTEM SystemLiteral NDATA NotationName>
```

*NotationName* is the name of a notation declared elsewhere in the DTD.

```
<?xml version="1.0"?>
<!DOCTYPE BOOK [
  <!ELEMENT BOOK (TITLE, AUTHOR, COVERIMAGE)>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT AUTHOR (#PCDATA)>
  <!ELEMENT COVERIMAGE EMPTY>
  <!ATTLIST COVERIMAGE Source ENTITY #REQUIRED>
  <!NOTATION GIF SYSTEM "">
  <!ENTITY faun SYSTEM "faun.gif" NDATA GIF>
]>
<BOOK>
  <TITLE></TITLE><AUTHOR></AUTHOR>
  <COVERIMAGE Source="faun" />
</BOOK>
```

An external unparsed entity is not accessed directly by the XML processor. Rather, the processor merely makes the entity and its notation available to the application, which can do what it wants with the information.

## DECLARING A NOTATION

A notation describes a particular data format. It does this by providing the address of a description of the format, the address of a program that can handle data in that format, or just a simple format description.

A notation has the following general form:

```
<!NOTATION NotationName SYSTEM SystemLiteral>
```

The *SystemLiteral* can include any format description that would be meaningful to the application that is going to display or handle the XML document. You might include there:

- the URI of a program that can process or display the data format, e.g.  

```
<!NOTATION BMP SYSTEM "Pbrush.exe">
<!NOTATION GIF SYSTEM "http://bogus.com/ShowGif.exe">
```
- the URI of an online document that describes the format, e.g.  

```
<!NOTATION STRANGIFORMAT SYSTEM "http://bogus.com/StrangeFormat.html">
```
- a simple description of the format, e.g.  

```
<!NOTATION GIF SYSTEM "Graphic Interchange Format">
```

## DECLARING A PARAMETER INTERNAL PARSED ENTITY

A parameter internal parsed entity uses the following general form:

```
<!ENTITY % EntityName EntityValue>
```

- *EntityName* follows the same rules applied to other entity names.

- You can insert a parameter entity only where a markup declaration can occur in the DTD, not within a markup declaration. Therefore, the *EntityValue* string must contain one or more complete markup declarations.

```
<?xml version="1.0"?>
<!DOCTYPE BOOK [
<!ENTITY % author
    "<!ELEMENT AUTHOR (#PCDATA)>
    <!ATTLIST AUTHOR Nationality CDATA 'American'>" >
<!ELEMENT BOOK (TITLE, AUTHOR)>
<!ELEMENT TITLE (#PCDATA)>
%author;
]>
<BOOK>
    <TITLE></TITLE><AUTHOR></AUTHOR>
</BOOK>
```

## DECLARING A PARAMETER EXTERNAL PARSED ENTITY

A parameter external parsed entity uses the following general form:

```
<!ENTITY % EntityName SYSTEM SystemLiteral>
```

The *SystemLiteral* specifies the URI of the file containing the parameter entity data.

```
<!DOCTYPE BOOK [
    <!ENTITY % author SYSTEM "author.ent">
    <!ELEMENT BOOK (TITLE, AUTHOR)>
    <!ELEMENT TITLE (#PCDATA)>
    %author;
]>
```

This facility is most frequently used to modularize a DTD.

```
<!DOCTYPE INVENTORY [
    <!ELEMENT INVENTORY (BOOK | CD)*>
    <!ENTITY % book.mod SYSTEM "book.mod">
    <!ENTITY % cd.mod SYSTEM "cd.mod">
    %book.mod;
    %cd.mod;
]>
```

Here are the contents of the book.mod file:

```
<!ELEMENT BOOK (BOOKTITLE, AUTHOR, PAGES)>
<!ELEMENT BOOKTITLE (#PCDATA)>
<!ELEMENT AUTHOR (#PCDATA)>
<!ELEMENT PAGES (#PCDATA)>
```

Here are the contents of the cd.mod file:

```
<!ELEMENT CD (CDTITLE, COMPOSER, DURATION)>
<!ELEMENT CDTITLE (#PCDATA)>
<!ELEMENT COMPOSER (#PCDATA)>
<!ELEMENT LENGTH (#PCDATA)>
```

## INSERTING CHARACTER REFERENCES

Use a character reference to insert a character that isn't on your keyboard, e.g. ö, or to insert a character that would be illegal in the current context, e.g. < or & within an element's character data. A character reference has two different forms:

- decimal -- &#n; where n is one or more decimal digits (0-9)
- hexadecimal -- &#xh; where h is one or more hexadecimal digits (0-F)

Both &#65; and &#x41; insert the capital letter A from the ISO/IEC 10646 character set (Unicode). Unicode is an international character set which contains characters from virtually every written language.

## USING PREDEFINED ENTITIES

XML has five predefined entities:

<u>Entity reference</u>	<u>Character inserted</u>	<u>Equivalent character reference</u>
&amp;	&	&#38;
&lt;	<	&#60;
&gt;	>	&#62;
&apos;	'	&#39;
&quot;	"	&#34;

## ADDING ENTITIES TO A DOCUMENT

1. Open inventory.dtd in a text editor.

2. Add the following general internal parsed entities:

```
<!-- entities for assigning to the BINDING element -->
<!ENTITY mass "mass market paperback">
<!ENTITY trade "trade paperback">
<!ENTITY hard "hardcover">
```

3. Add the following external unparsed entities:

```
<!-- external entities containing reviews -->
<!NOTATION DOC SYSTEM "Microsoft Word document">
<!NOTATION TXT SYSTEM "ASCII text file">
<!NOTATION HTML SYSTEM "HTML">
<!ENTITY rev-leaves SYSTEM "rev-leaves.doc" NDATA DOC>
<!ENTITY rev-faun1 SYSTEM "rev-faun.doc" NDATA DOC>
<!ENTITY rev-faun2 SYSTEM "rev-faun.txt" NDATA TXT>
<!ENTITY rev-faun3 SYSTEM "rev-faun.html" NDATA HTML>
<!ENTITY rev-screw SYSTEM "rev-screw.txt" NDATA TXT>
<!ENTITY rev-sleepy SYSTEM "rev-sleepy.html" NDATA HTML>
```

4. Edit the attribute-list declaration for the BOOK element so that it contains:

```
<!ATTLIST BOOK      InStock      (yes|no)      #REQUIRED
                   Reviews        ENTITIES      #IMPLIED>
```

5. In each BINDING element, replace the content with the corresponding entity reference, e.g.

```
<BINDING>mass market paperback</BINDING> becomes <BINDING>&mass;</BINDING>
```

6. Add Reviews attributes to the appropriate BOOK elements.

## DISPLAYING XML USING XSL STYLE SHEETS

While CSS merely allows you to specify the formatting of each XML element, an XSL style sheet gives you complete control over the output. XSL allows you to --

- access all XML components, such as elements, attributes, comments, and processing instructions
- select the XML data you want to display
- present that data in any order or arrangement, e.g. sort and filter
- freely modify or add information

XSL style sheets can transform XML to HTML which the browser then renders, giving you access to the full formatting and functional richness of HTML, in addition to the data access and transforming features provided by XSL itself.

The XSL style sheet is linked to the XML document via a processing instruction, which has the following general form:

```
<?xml-stylesheet type="text/xsl" href="[XSL URI]" ?>
```

Style sheets must have the following top-level document element:

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

An XSL style sheet includes one or more templates, each of which contains the information for displaying a particular branch of the XML document tree. They take the following general form:

```
<xsl:template match="/">
    <!-- child elements -->
</xsl:template>
```

The match attribute indicates the specific branch that this template should be applied to. The value of the match attribute is known as a pattern. The pattern in the example above represents the root of the entire XML document. Every XSL style sheet must contain exactly one template with the match attribute set to the value "/". It's important to realize that the root pattern does NOT represent the document element of the XML document. Instead, it represents the entire document, of which the document element is a child.

A template contains two kinds of XML elements:

- XML elements that represent HTML markup, e.g. <H2>Book Description</H2>
- XSL elements, e.g. <xsl:value-of select="INVENTORY/BOOK/AUTHOR" />

Each of the elements representing HTML markup must be a well-formed XML element as well. Therefore, you can't use HTML constructs such as an element with only a start-tag. For example, to specify an HTML line break element, you must use the XML empty-element tag <BR />. HTML elements are copied to the output.

XSL elements are distinguished from HTML elements because they have the xsl namespace designation, e.g. xsl:template. XSL elements are NOT copied to the output. Instead, they contain instructions for selecting or modifying the XML data and performing other tasks.

```

<?xml version="1.0"?>
<!-- File Name: demol.xsl -->

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <HTML>
      <H2>Book Description</H2>
      <SPAN STYLE="font-style:italic">Author: </SPAN>
      <xsl:value-of select="INVENTORY/BOOK/AUTHOR"/><BR />
      <SPAN STYLE="font-style:italic">Title: </SPAN>
      <xsl:value-of select="INVENTORY/BOOK/TITLE"/><BR />
      <SPAN STYLE="font-style:italic">Price: </SPAN>
      <xsl:value-of select="INVENTORY/BOOK/PRICE"/><BR />
      <SPAN STYLE="font-style:italic">Binding type: </SPAN>
      <xsl:value-of select="INVENTORY/BOOK/BINDING"/><BR />
      <SPAN STYLE="font-style:italic">Number of pages: </SPAN>
      <xsl:value-of select="INVENTORY/BOOK/PAGES"/><BR />
    </HTML>
  </xsl:template>
</xsl:stylesheet>

```

Why does this style sheet only display the first book element?

## DISPLAYING A VARIABLE NUMBER OF ELEMENTS WITH A FOR-EACH LOOP

```

<?xml version="1.0"?>
<!-- File Name: demo2.xsl -->

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <HTML>
      <H2>Book Description</H2>
      <xsl:for-each select="INVENTORY/BOOK">
        <SPAN STYLE="font-style:italic">Author: </SPAN>
        <xsl:value-of select="AUTHOR"/><BR />
        <SPAN STYLE="font-style:italic">Title: </SPAN>
        <xsl:value-of select="TITLE"/><BR />
        <SPAN STYLE="font-style:italic">Price: </SPAN>
        <xsl:value-of select="PRICE"/><BR />
        <SPAN STYLE="font-style:italic">Binding type: </SPAN>
        <xsl:value-of select="BINDING"/><BR />
        <SPAN STYLE="font-style:italic">Number of pages: </SPAN>
        <xsl:value-of select="PAGES"/><BR /><BR />
      </xsl:for-each>
    </HTML>
  </xsl:template>
</xsl:stylesheet>

```

The for-each element has two main effects:

- The output from the block of elements contained within the for-each element repeats once for every XML element that matches the pattern assigned to the for-each element's select attribute.
- Within the for-each element, the current element is the element specified by the for-each element's select attribute.

The final result is that the output contains the data from all the BOOK elements found in the INVENTORY element, regardless of how many there are.

## DISPLAYING A VARIABLE NUMBER OF ELEMENTS WITH MULTIPLE TEMPLATES

Another way to display a repeated XML element is to create a separate template for that element, and then invoke that template using the XSL apply-templates element.

```
<?xml version="1.0"?>
<!-- File Name: demo3.xsl -->

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <HTML>
    <H2>Book Description</H2>
    <xsl:apply-templates select="INVENTORY/BOOK" />
    </HTML>
  </xsl:template>

  <xsl:template match="BOOK">
    <SPAN STYLE="font-style:italic">Author: </SPAN>
    <xsl:value-of select="AUTHOR"/><BR />
    <SPAN STYLE="font-style:italic">Title: </SPAN>
    <xsl:value-of select="TITLE"/><BR />
    <SPAN STYLE="font-style:italic">Binding type: </SPAN>
    <xsl:value-of select="BINDING"/><BR />
    <SPAN STYLE="font-style:italic">Number of pages: </SPAN>
    <xsl:value-of select="PAGES"/><BR />
    <SPAN STYLE="font-style:italic">Price: </SPAN>
    <xsl:value-of select="PRICE"/><BR /><BR />
  </xsl:template>
</xsl:stylesheet>
```

## FILTERING XML DATA USING STYLE SHEETS

The patterns you've seen so far have contained only a path operator that specifies the element's name and possibly one or more of its ancestor elements. You can further restrict the number of elements that the pattern matches by including a filter surrounded by square brackets immediately following the path operator. Simply including an element name in the filter indicates that a matching element must have a child with the included name. For example, the following pattern matches any ITEM element that has a child element named CD, regardless of the contents of CD:

```
<xsl:template match="ITEM[CD]">
```

The following pattern matches any SHIRT element that has a child COLOR element matching (not containing!) the text "red":

```
<xsl:template match="SHIRT[COLOR='red']">
```

What will happen if you use the following?

```
<xsl:apply-templates select="INVENTORY/BOOK[BINDING='trade paperback']" />
```

## SORTING XML DATA USING STYLE SHEETS

The `xsl:sort` element can be used inside `xsl:for-each` and `xsl:apply-templates` to determine the order in which elements are processed and, therefore, displayed.

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

<!-- File Name: demo4.xsl -->

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <HTML>
      <H2>Book Description</H2>
      <xsl:apply-templates select="INVENTORY/BOOK[BINDING='trade
paperback']">
        <xsl:sort select="TITLE"/>
      </xsl:apply-templates>
    </HTML>
  </xsl:template>

  <xsl:template match="BOOK">
<SPAN STYLE="font-style:italic">Author: </SPAN>
  <xsl:value-of select="AUTHOR"/><BR />
<SPAN STYLE="font-style:italic">Title: </SPAN>
  <xsl:value-of select="TITLE"/><BR />
<SPAN STYLE="font-style:italic">Binding type: </SPAN>
  <xsl:value-of select="BINDING"/><BR />
<SPAN STYLE="font-style:italic">Number of pages: </SPAN>
  <xsl:value-of select="PAGES"/><BR />
<SPAN STYLE="font-style:italic">Price: </SPAN>
  <xsl:value-of select="PRICE"/><BR /><BR />
  </xsl:template>
</xsl:stylesheet>
```

## ACCESSING XML ATTRIBUTES

XSL treats an attribute belonging to an element in the XML document as if it were a child element. However, to reference the attribute in an XSL pattern, you must preface the attribute's name with the `@` character, which indicates that the name refers to an attribute rather than to an element.

The following filter selects all `BOOK` elements with an attributed named `InStock` that has the value "yes".

```
<xsl:apply-templates select="INVENTORY/BOOK[@InStock='yes']">
```

You can use the `value-of` element to extract the value of an attribute in the same way you use it to extract the text content of an element. The following `value-of` element obtains the value of the `Born` attribute belonging to the `AUTHOR` element:

```
<xsl:value-of select="AUTHOR/@Born" />
```

## **PUTTING IT ALL TOGETHER**

1. Write a style sheet that displays the contents of inventory.xml as an HTML table.
2. Modify the style sheet to select only those books that are in stock.
3. Display the in stock books sorted by author.

## BIBLIOGRAPHY

### Introductory --

- Simpson, John E. Just XML. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001.  
Young, Michael J. XML Step by Step. 2nd ed. Redmond, WA: Microsoft Press, 2002.

### Intermediate --

- Bradley, Neil. XML Companion. 3rd ed. New York: Addison-Wesley, 2002.  
Harold, Elliotte Rusty. XML Bible. 2nd ed. Foster City, CA: IDG Books Worldwide, 2001.  
St. Laurent, Simon. XML Elements of Style. New York: McGraw-Hill, 2000.

### Advanced --

- Anderson, Richard, et. al. Professional XML. Birmingham, UK: Wrox Press, 2000.  
Kay, Michael. XSLT Programmer's Reference. Birmingham, UK: Wrox Press, 2000.  
Maler, Eve and Jeanne El Andaloussi. Developing SGML DTDs: From Text to Model to Markup. Upper Saddle River, NJ: Prentice-Hall, 1996.  
Unicode Consortium. Unicode Standard: Version 3.0. Reading, MA: Addison-Wesley, 2000.  
XSL Transformations (XSLT). Version 1.0. W3C Recommendation 16 November 1999.  
(<http://www.w3.org/TR/xslt.html>).