

# Embedding Non-XML Data

---

**N**ot all data in the world is XML. In fact, I'd venture to say that most of the world's accumulated data isn't XML. A heck of a lot is stored in plain text, HTML, and Microsoft Word—to name just three common non-XML formats. And while most of this data could at least in theory be rewritten as XML—interest and resources permitting—not all of the world's data should be XML. Encoding images in XML, for example, would be extremely inefficient.

XML provides three constructs generally used for working with non-XML data: notations, unparsed external entities, and processing instructions. Notations describe the format of non-XML data. Unparsed external entities provide links to the actual location of the non-XML data. Processing instructions give information about how to view the data.



The material discussed in this chapter is very controversial. Although everything I describe is part of the XML 1.0 specification, not everyone agrees that it should be. You can certainly write XML documents without using any notations or unparsed external entities, and with only a few simple processing instructions. You may want to skip over this chapter at first, and return later if you discover a need for it.

## Notations

The first problem you encounter when working with non-XML data in an XML document is identifying the format of the data and telling the XML application how to read and display the non-XML data. For example, it would be inappropriate to try to draw an MP3 sound file on the screen.

To a limited extent, you can solve this problem within a single application by using only a fixed set of tags for particular



### In This Chapter

Notations

Unparsed external entities

Processing instructions

Conditional sections in DTDs



kinds of external entities. For instance, if all pictures are embedded through `IMAGE` elements and all sounds via `AUDIO` elements, then it's not hard to develop a browser that knows how to handle those two elements. In essence, this is the approach that HTML takes. However, this approach does prevent document authors from creating new tags that more specifically describe their content; for example, a `PERSON` element that happens to have a `PHOTO` attribute that points to a JPEG image of that person.

Furthermore, no application understands all possible file formats. Most Web browsers can recognize and read GIF, JPEG, PNG-and perhaps a few other kinds of image files-but they fail completely when faced with EPS files, TIFF files, FITS files, or any of the hundreds of other common and uncommon image formats. The dialog in Figure 11-1 is probably all too familiar.



**Figure 11-1:** What occurs when Netscape Navigator doesn't recognize a file type

Ideally, you want documents to tell the application the format of the external entity so you don't have to rely on the application recognizing the file type by a magic number or a potentially unreliable file name extension. Furthermore, you'd like to give the application some hints about what program it can use to display the image if it's unable to do so itself.

Notations provide a partial (although not always well supported) solution to this problem. Notations describe the format of non-XML data. A `NOTATION` declaration in the DTD specifies a particular data type. The DTD declares notations at the same level as elements, attributes, and entities. Each notation declaration contains a name and an external identifier according to the following syntax:

```
<!NOTATION name SYSTEM "externalID">
```

The *name* is an identifier for this particular format used in the document. The *externalID* contains a human intelligible string that somehow identifies the notation. For instance, you might use MIME types like those used in this notation for GIF images:

```
<!NOTATION GIF SYSTEM "image/gif">
```

You can also use a PUBLIC identifier instead of the SYSTEM identifier. To do this, you must provide both a public ID and a URL. For example,

```
<!NOTATION GIF PUBLIC
  "-//IETF//NONSGML Media Type image/gif//EN"
  "http://www.isi.edu/in-notes/iana/assignments/media-
  types/image/gif">
```



Caution

There is *a lot* of debate about what exactly makes a good external identifier. MIME types like image/gif or text/html are one possibility. Another suggestion is URLs or other locators for standards documents like <http://www.w3.org/TR/REC-html40/>. A third option is the name of an official international standard like ISO 8601 for representing dates and times. In some cases, an ISBN or Library of Congress catalog number for the paper document where the standard is defined might be more appropriate. And there are many more choices.

Which you choose may depend on the expected life span of your document. For instance, if you use an unusual format, you don't want to rely on a URL that changes from month to month. If you expect your document to still spark interest in 100 years, then you may want to consider which identifiers are likely to still have meaning in 100 years and which are merely this decade's technical ephemera.

You can also use notations to describe data that does fit in an XML document. For instance, consider this DATE element:

```
<DATE>05-07-06</DATE>
```

What day, exactly, does 05-07-06 represent? Is it May 7, 1906 C.E.? Or is it July 5, 1906 C.E.? The answer depends on whether you read this in the United States or Europe. Maybe it's even May 7, 2006 C.E. or July 5, 2006 C.E. Or perhaps what's meant is May 7, 6 C.E., during the reign of the Roman emperor Augustus in the West and the Han dynasty in China. It's also possible that date isn't in the "Common Era" at all but is given in the traditional Jewish, Muslim, or Chinese calendars. Without more information, you cannot determine the true meaning.

To avoid confusion like this, ISO standard 8601 defines a precise means of representing dates. In this scheme, July 5, 2006 C.E. is written as 20060705 or, in XML, as follows:

```
<DATE>20060705</DATE>
```

This format doesn't match *anybody's* expectations; it's equally confusing to everybody and thus has the advantage of being more or less culturally neutral (though still biased toward the traditional Western calendar).

Notations are declared in the DTD and used in notation attributes to describe the format of non-XML data embedded in an XML document. To continue with the date example, Listing 11-1 defines two possible notations for dates in ISO 8601 and conventional U.S. formats. Then, a required `FORMAT` attribute of type `NOTATION` is added to each `DATE` element to describe the structure of the particular element.

### Listing 11-1: DATE elements in an ISO 8601 and conventional U.S. formats

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SCHEDULE [

    <!NOTATION ISODATE SYSTEM
        "http://www.iso.ch/cate/d15903.html">
    <!NOTATION USDATE SYSTEM
        "http://es.rice.edu/ES/humsoc/Galileo/Things/gregorian_calendar
        .html">

    <!ELEMENT SCHEDULE (APPOINTMENT*)>
    <!ELEMENT APPOINTMENT (NOTE, DATE, TIME?)>

    <!ELEMENT NOTE (#PCDATA)>
    <!ELEMENT DATE (#PCDATA)>
    <!ELEMENT TIME (#PCDATA)>

    <!ATTLIST DATE FORMAT NOTATION (ISODATE | USDATE) #IMPLIED>

]>
<SCHEDULE>
  <APPOINTMENT>
    <NOTE>Deliver presents</NOTE>
    <DATE FORMAT="USDATE">12-25-1999</DATE>
  </APPOINTMENT>
  <APPOINTMENT>
    <NOTE>Party like it's 1999</NOTE>
    <DATE FORMAT="ISODATE">19991231</DATE>
  </APPOINTMENT>
</SCHEDULE>
```

Notations can't force authors to use the format described by the notation. For that you need to use some sort of schema language in addition to basic XML—but it is sufficient for simple uses where you trust authors to correctly describe their data.

## Unparsed External Entities

XML is not an ideal format for all data, particularly non-text data. For instance, you could store each pixel of a bitmap image as an XML element, as shown below:

```
<PIXEL X="232" Y="128" COLOR="FF5E32" />
```

This is hardly a good idea, though. Anything remotely like this would cause your image files to balloon to obscene proportions. Since you can't encode all data in XML, XML documents must be able to refer to data not currently XML and probably never will be.

A typical Web page may include GIF and JPEG images, Java applets, ActiveX controls, various kinds of sounds, and so forth. In XML, any block of non-XML data is called an *unparsed entity* because the XML processor won't attempt to understand it. At most, it informs the application of the entity's existence and provides the application with the entity's name and possibly (though not necessarily) its content.

HTML pages embed non-HTML entities through a variety of custom tags. Pictures are included with the `<IMG>` tag whose `SRC` attribute provides the URL of the image file. Applets are embedded via the `<APPLET>` tag whose `CLASS` and `CODEBASE` attributes refer to the file and directory where the applet resides. The `<OBJECT>` tag uses its `codebase` attribute to refer to the URI from where the object's data is found. In each case, a particular predefined tag represents a particular kind of content. A predefined attribute contains the URL for that content.

XML applications can work like this, but they don't have to. In fact, most don't unless they're deliberately trying to maintain some level of backwards compatibility with HTML. Instead, XML applications use an unparsed external entity to refer to the content. Unparsed external entities provide links to the actual location of the non-XML data. Then they use an `ENTITY` type attribute to associate that entity with a particular element in the document.

## Declaring Unparsed Entities

Recall from Chapter 9 that an external entity declaration looks something like this:

```
<!ENTITY SIG SYSTEM "http://metalab.unc.edu/xml/signature.xml">
```

However, this form is only acceptable if the external entity the URL names is more or less a well-formed XML document. If the external entity is not XML, then you have to specify the entity's type using the `NDATA` keyword. For example, to associate the GIF file `logo.gif` with the name `LOGO`, you would place this `ENTITY` declaration in the DTD:

```
<!ENTITY LOGO SYSTEM "logo.gif" NDATA GIF>
```

The final word in the declaration, GIF in this example, must be the name of a notation declared in the DTD. Notations associate a name like GIF with some sort of external identifier for the format such as a MIME type, an ISO standard, or the URL of a specification of the format. For example, the notation for GIF might look like this:

```
<!NOTATION GIF SYSTEM "image/gif">
```

As usual, you can use absolute or relative URLs for the external entity as convenience dictates. For example,

```
<!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
  NDATA GIF>
<!ENTITY LOGO SYSTEM "/xml/logo.gif" NDATA GIF>
<!ENTITY LOGO SYSTEM "../logo.gif" NDATA GIF>
```

## Embedding Unparsed Entities

You cannot simply embed an unparsed entity at an arbitrary location in the document using a general entity reference as you can with parsed entities. For instance, Listing 11-2 is an invalid XML document because LOGO is an unparsed entity. If LOGO were a parsed entity, this example would be valid.

### Listing 11-2: An invalid XML document that tries to embed an unparsed entity with a general entity reference

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT ANY>
  <!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
    NDATA GIF>
  <!NOTATION GIF SYSTEM "image/gif">
]>
<DOCUMENT>
  &LOGO;
</DOCUMENT>
```

To embed unparsed entities, rather than using general entity references like &LOGO;, you declare an element that serves as a placeholder for the unparsed entity (IMAGE, for example). Then you declare an ENTITY type attribute for the IMAGE element-SOURCE, for example-which provides only the name of the unparsed entity. Listing 11-3 demonstrates.

### Listing 11-3: A valid XML document that correctly embeds an unparsed entity

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [

  <!ELEMENT DOCUMENT ANY>
  <!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
    NDATA GIF>
  <!NOTATION GIF SYSTEM "image/gif">
  <!ELEMENT IMAGE EMPTY>
  <!ATTLIST IMAGE SOURCE ENTITY #REQUIRED>

]>
<DOCUMENT>
  <IMAGE SOURCE="LOGO" />
</DOCUMENT>

```

It is now up to the application reading the XML document to recognize the unparsed entity and display it. Applications may not display the unparsed entity (just as a Web browser may choose not to load images when the user has disabled image loading).

These examples show empty elements as the containers for unparsed entities. That's not always necessary, however. For instance, imagine an XML-based corporate ID system that a security guard uses to look up people entering a building. The PERSON element might have NAME, PHONE, OFFICE, and EMPLOYEE\_ID children and a PHOTO ENTITY attribute. Listing 11-4 demonstrates.

### Listing 11-4: A non-empty PERSON element with a PHOTO ENTITY attribute

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE PERSON [
  <!ELEMENT PERSON (NAME, EMPLOYEE_ID, PHONE, OFFICE)>
  <!ELEMENT NAME (#PCDATA)>
  <!ELEMENT EMPLOYEE_ID (#PCDATA)>
  <!ELEMENT PHONE (#PCDATA)>
  <!ELEMENT OFFICE (#PCDATA)>
  <!NOTATION JPEG SYSTEM "image/jpg">
  <!ENTITY ROGER SYSTEM "rogers.jpg" NDATA JPEG>

```

*Continued*

## Listing 11-4 (continued)

```

<!ATTLIST PERSON PHOTO ENTITY #REQUIRED>

]>
<PERSON PHOTO="ROGER">
  <NAME>Jim Rogers</NAME>
  <EMPLOYEE_ID>4534</EMPLOYEE_ID>
  <PHONE>X396</PHONE>
  <OFFICE>RH 415A</OFFICE>
</PERSON>

```

This example may seem a little artificial. In practice, you'd be better advised to make an empty PHOTO element with a SOURCE attribute a child of a PERSON element rather than an attribute of PERSON. Furthermore, you'd probably separate the DTD into external and internal subsets. The external subset, shown in Listing 11-5, declares the elements, notations, and attributes. These are the parts likely to be shared among many different documents. The entity, however, changes from document to document. Thus, you can better place it in the internal DTD subset of each document as shown in Listing 11-6.

## Listing 11-5: The external DTD subset person.dtd

```

<!ELEMENT PERSON (NAME, EMPLOYEE_ID, PHONE, OFFICE, PHOTO)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT EMPLOYEE_ID (#PCDATA)>
<!ELEMENT PHONE (#PCDATA)>
<!ELEMENT OFFICE (#PCDATA)>
<!ELEMENT PHOTO EMPTY>
<!NOTATION JPEG SYSTEM "image/jpeg">
<!ATTLIST PHOTO SOURCE ENTITY #REQUIRED>

```

## Listing 11-6: A document with a non-empty PERSON element and an internal DTD subset

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE PERSON [
  <!ENTITY % PERSON_DTD SYSTEM "person.dtd">
  %PERSON_DTD;
  <!ENTITY ROGER SYSTEM "rogers.jpg" NDATA JPEG>

```

```

]>
<PERSON>
  <NAME>Jim Rogers</NAME>
  <EMPLOYEE_ID>4534</EMPLOYEE_ID>
  <PHONE>X396</PHONE>
  <OFFICE>RH 415A</OFFICE>
  <PHOTO SOURCE="ROGER"/>
</PERSON>

```

---

## Embedding Multiple Unparsed Entities

On rare occasions, you may need to refer to more than one unparsed entity in a single attribute, perhaps even an indefinite number. You can do this by declaring an attribute of the entity placeholder to have type `ENTITIES`. An `ENTITIES` type attribute has a value part that consists of multiple unparsed entity names separated by white space. Each entity name refers to an external non-XML data source and must be declared in the DTD. For example, you might use this to write a slide show element that rotates different pictures. The DTD would require these declarations:

```

<!ELEMENT SLIDESHOW EMPTY>
<!ATTLIST SLIDESHOW SOURCES ENTITIES #REQUIRED>
<!NOTATION JPEG SYSTEM "image/jpeg">
<!ENTITY CHARM SYSTEM "charm.jpg" NDATA JPEG>
<!ENTITY MARJORIE SYSTEM "marjorie.jpg" NDATA JPEG>
<!ENTITY POSSUM SYSTEM "possum.jpg" NDATA JPEG>
<!ENTITY BLUE SYSTEM "blue.jpg" NDATA JPEG>

```

Then, at the point in the document where you want the slide show to appear, you insert the following tag:

```
<SLIDESHOW SOURCES="CHARM MARJORIE POSSUM BLUE">
```

Once again, I must emphasize that this is not a magic formula that all (or even any) XML browsers automatically understand. It is simply one technique browsers and other applications may or may not adopt to embed non-XML data in documents.

## Processing Instructions

Comments often get abused to support proprietary extensions to HTML like server side includes, browser-specific scripting languages, database templates, and several dozen other items outside the purview of the HTML standard. The advantage of using comments for these purposes is that other systems simply ignore the extraneous data they don't understand. The disadvantage of this

approach is that a document stripped of its comments may no longer be the same document, and that comments intended as mere documentation may be unintentionally processed as input to these proprietary extensions. To avoid this abuse of comments, XML provides the *processing instruction*—an explicit mechanism for embedding information in a file intended for proprietary applications rather than the XML parser or browser. Among other uses, processing instructions can provide additional information about how to view unparsed external entities.

A processing instruction is a string of text between `<? and ?>` marks. The only required syntax for the text inside the processing instruction is that it must begin with an XML name followed by white space followed by data. The XML name may either be the actual name of the application (e.g., `latex`) or the name of a notation in the DTD that points to the application (e.g., `LATEX`) where `LATEX` is declared like this in the DTD:

```
<!NOTATION LATEX SYSTEM "/usr/local/bin/latex">
```

It may even be a name that is recognized by an application with a different name. The details tend to be very specific to the application for which the processing instruction is intended. Indeed, most applications that rely on processing instructions will impose more structure on the contents of a processing instruction. For example, consider this processing instruction used in IBM's Bean Markup Language:

```
<?bmlpi register demos.calculator.EventSourceText2Int?>
```

The name of the application this instruction is intended for is `bmlpi`. The data given to that application is the string `register demos.calculator.EventSourceText2Int`, which happens to include the full package qualified name of a Java class. This tells the application named `bmlpi` to use the Java class `demos.calculator.EventSourceText2Int` to convert action events to integers. If `bmlpi` encounters this processing instruction while reading the document, it will load the class `demos.calculator.EventSourceText2Int` and use it to convert events to integers from that point on.

If this sounds fairly specific and detailed, that's because it is. Processing instructions are not part of the general structure of the document. They are intended to provide extra, detailed information for particular applications, not for every application that reads the document. If some other application encounters this instruction while reading a document, it will simply ignore the instruction.

Processing instructions may be placed almost anywhere in an XML document except inside a tag or a `CDATA` section. They may appear in the prolog or the

DTD, in the content of an element, or even after the closing document tag. Since processing instructions are not elements, they do not affect the tree structure of a document. You do not need to open or close processing instructions, or worry about how they nest inside other elements. Processing instructions are not tags and they do not delimit elements.

You're already familiar with one example of processing instructions, the `xml-stylesheet` processing instruction used to bind style sheets to documents:

```
<?xml-stylesheet type="text/xsl" href="baseball.xsl"?>
```

Although these examples appear in a document's prolog, in general processing instructions may appear anywhere in a document. You do not need to declare these instructions as child elements of the element they are contained in because they're not elements.

Processing instructions that begin with the string `xml` are reserved for uses defined in the XML standard. Otherwise, you are free to use any name and any string of text inside a processing instruction other than the closing string `?>`. For instance, the following examples are all valid processing instructions:

```
<?gcc HelloWorld.c ?>  
<?acrobat document="passport.pdf"?>  
<?Dave remember to replace this one?>
```

Note

Remember an XML processor won't necessarily do anything with these instructions. It merely passes them along to the application. The application decides what to do with the instructions. Most applications simply ignore processing instructions they don't understand.

Sometimes knowing the type of an unparsed external entity is insufficient. You may also need to know what program to run to view the entity and what parameters you need to provide that program. You can use a processing instruction to provide this information. Since processing instructions can contain fairly arbitrary data, it's relatively easy for them to contain instructions determining what action the external program listed in the notation should take.

Such a processing instruction can range from simply the name of a program that can view the file to several kilobytes of configuration information. The application and the document author must of course use the same means of determining which processing instructions belong with which unparsed external entities. Listing 11-7 shows one scheme that uses a processing instruction and a PDF notation to try to pass the PDF version of a physics paper to Acrobat Reader for display.

## Listing 11-7: Embedding a PDF document in XML

```

<?xml version="1.0" standalone="yes"?>
<!DOCTYPE PAPER [

    <!NOTATION PDF PUBLIC
        "-//IETF//NONSGML Media Type application/pdf//EN"
        "http://www.isi.edu/in-notes/iana/assignments/media-
types/application/pdf">

    <!ELEMENT PAPER (TITLE, AUTHOR+, JOURNAL, DATE_RECEIVED,
VOLUME, ISSUE, PAGES)>
    <!ATTLIST PAPER CONTENTS ENTITY #IMPLIED>
    <!ENTITY PRLTA0000081000024005270000001 SYSTEM

"http://ojps.aip.org/journal_cgi/getpdf?KEY=PRLTA0&cvips=PR
LTA0000081000024005270000001"
    NDATA PDF>

    <!ELEMENT AUTHOR (#PCDATA)>
    <!ELEMENT JOURNAL (#PCDATA)>
    <!ELEMENT YEAR (#PCDATA)>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT DATE_RECEIVED (#PCDATA)>
    <!ELEMENT VOLUME (#PCDATA)>
    <!ELEMENT ISSUE (#PCDATA)>
    <!ELEMENT PAGES (#PCDATA)>

]>

<?PDF acroread?>
<PAPER CONTENTS="PRLTA0000081000024005270000001">
  <TITLE>Do Naked Singularities Generically Occur in
Generalized Theories of Gravity?</TITLE>
  <AUTHOR>Kengo Maeda</AUTHOR>
  <AUTHOR>Takashi Torii</AUTHOR>
  <AUTHOR>Makoto Narita</AUTHOR>
  <JOURNAL>Physical Review Letters</JOURNAL>
  <DATE_RECEIVED>19 August 1998</DATE_RECEIVED>
  <VOLUME>81</VOLUME>
  <ISSUE>24</ISSUE>
  <PAGES>5270-5273</PAGES>
</PAPER>

```

---

As always, you have to remember that not every processor will treat this example in the way intended. In fact, most won't. However, this is one possible scheme for how an application might support PDF files and other non-XML media types.

## Conditional Sections in DTDs

When developing DTDs or documents, you may need to comment out parts of the DTD not yet reflected in the documents. In addition to using comments directly, you can omit a particular group of declarations in the DTD by wrapping it in an `IGNORE` directive. The syntax follows:

```
<![ IGNORE
  declarations that are ignored
  ]>
```

As usual, white space doesn't really affect the syntax, but you should keep the opening `<![ IGNORE` and the closing `]>` on separate lines for easy viewing.

You can ignore any declaration or combination of declarations — elements, entities, attributes, or even other `IGNORE` blocks — but you must ignore entire declarations. The `IGNORE` construct must completely enclose the entire declarations it removes from the DTD. You cannot ignore a piece of a declaration (such as the `CDATA` `GIF` in an unparsed entity declaration).

You can also specify that a particular section of declarations is included — that is, not ignored. The syntax for the `INCLUDE` directive is just like the `IGNORE` directive but with a different keyword:

```
<![ INCLUDE
  declarations that are included
  ]>
```

When an `INCLUDE` is inside an `IGNORE`, the `INCLUDE` and its declarations are ignored. When an `IGNORE` is inside an `INCLUDE`, the declarations inside the `IGNORE` are still ignored. In other words, an `INCLUDE` never overrides an `IGNORE`.

Given these conditions, you may wonder why `INCLUDE` even exists. No DTD would change if all `INCLUDE` blocks were simply removed, leaving only their contents. `INCLUDE` appears to be completely extraneous. However, there is one neat trick with parameter entity references and both `IGNORE` and `INCLUDE` that you can't do with `IGNORE` alone. First, define a parameter entity reference as follows:

```
<!ENTITY % fulltdtd "IGNORE">
```

You can ignore elements by wrapping them in the following construct:

```
<![ %fulltdtd;
  declarations
  ]>
```

The `%fulltd;` parameter entity reference evaluates to `IGNORE`, so the declarations are ignored. Now, suppose you make the one word edit to change `fulltd` from `IGNORE` to `INCLUDE` as follows:

```
<!ENTITY % fulltd "INCLUDE">
```

Immediately, all the `IGNORE` blocks convert to `INCLUDE` blocks. In effect, you have a one-line switch to turn blocks on or off.

In this example, I've only used one switch, `fulltd`. You can use this switch in multiple `IGNORE/INCLUDE` blocks in the DTD. You can also have different groups of `IGNORE/INCLUDE` blocks that you switch on or off based on different conditions.

You'll find this capability particularly useful when designing DTDs for inclusion in other DTDs. The ultimate DTD can change the behavior of the DTDs it embeds by changing the value of the parameter entity switch.

## Summary

In this chapter, you learned how to integrate your XML documents with non XML data through notations, unparsed external entities, and processing instructions. In particular, you learned the following concepts:

- ♦ Notations describe the type of non-XML data.
- ♦ Unparsed external entities are storage units containing non-XML text or binary data.
- ♦ Unparsed external entities are included in documents using `ENTITY` or `ENTITIES` attributes.
- ♦ Processing instructions contain instructions passed along unchanged from the XML processor to the ultimate application.
- ♦ `INCLUDE` and `IGNORE` blocks specify that the enclosed declarations of the DTD are or are not (respectively) to be considered when parsing the document.

You'll see a lot more examples of documents with DTDs over the next several parts of this book, but as far as basic syntax and usage goes, this chapter concludes the exploration of DTDs. In Part III, we begin discussion of style languages for XML, beginning in the next chapter with Cascading Style Sheets, Level 1.

