[Figures are not included in this sample chapter]

Sams Teach Yourself XML In 21 Days

- 3 -

Using XML Markup

Yesterday, in "Anatomy of an XML Document," you learned about the main features of XML markup for elements and entities. Today’s chapter will expand on this, and you’ll also learn about the following:

- Attributes
- Entity references and how to use them as shortcuts
- How to include comments in your code
- What CDATA sections are and how they are used
- Processing instructions

Markup Delimiters

Yesterday you learned about XML’s markup characters in fairly general terms. Now it’s time to get a little more technical and examine the exact details of XML’s markup declarations.

Table 3.1 identifies the parts of XML’s element tags. When the details get a bit more technical, it will be helpful if you’re familiar with these parts. (Although you don’t need to commit them to memory!)

**TABLE 3.1 THE PARTS OF AN ELEMENT TAG**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Start tag open delimiter</td>
</tr>
<tr>
<td>&lt;/</td>
<td>End tag open delimiter</td>
</tr>
<tr>
<td>foo</td>
<td>Element name</td>
</tr>
<tr>
<td>&gt;</td>
<td>Tag close delimiter</td>
</tr>
<tr>
<td>*/</td>
<td>Empty tag close delimiter</td>
</tr>
</tbody>
</table>

It is worth remembering that, whereas HTML simply relies on recognizing preprogrammed tags, XML is triggered by these specific parts of the element tags, and the XML processor’s behavior and what it expects to see next are directly controlled by the named symbols.
Element Markup

XML is concerned with element markup. This might sound like an obvious point to make, but it is worth repeating because it indicates a deeply rooted conceptual difference between XML as a markup language and an arbitrary tag language. As you have already seen, HTML often tends toward being a tag language rather than a markup language. This is a direct consequence of Web browsers being so intentionally lenient in accepting bad markup.

Instead of XML’s tags being markers that indicate where a style should change or a new line should begin, XML’s element markup is composed of three parts: a start tag, the contents, and the end tag. This is shown in Table 3.2. The start tag and end tag should be treated like wrappers, and when you think of an element, you should have a mental picture of a piece of text with both tags in place.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;foo&gt;</td>
<td>Start tag</td>
<td>At the start of an element, the opening tag</td>
</tr>
<tr>
<td>text</td>
<td>Content</td>
<td>In the middle of an element, its content</td>
</tr>
<tr>
<td>&lt;/foo&gt;</td>
<td>End tag</td>
<td>At the end of an element, the closing tag</td>
</tr>
</tbody>
</table>

Note that the element name that appears in the start tag must be exactly the same as the name that appears in the end tag. For example, the following would be wrong:

```xml
<simple.element>This element won’t close!</simple.Element>
```

CAUTION: The first versions of XML, before it became a full-blown proposal, were not case sensitive. There are still some XML software packages in circulation that are not case sensitive and will not signal an error if you mix up cases. For conformity with XML requirements, you must be careful to keep your use of upper- and lowercase consistent.

Attribute Markup

As you learned yesterday, attributes are used to attach information to the information contained in an element. The general form for using an attribute is

```xml
<!element.name property="value">
```

or

```xml
<!element.name property='value'>
```

The technical description of the markup of this attribute specification is given in Table 3.3.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Element.name</td>
<td>Property=&quot;value&quot;</td>
</tr>
<tr>
<td></td>
<td>Element.name</td>
<td>Property='value'</td>
</tr>
</tbody>
</table>

The technical description of the markup of this attribute specification is given in Table 3.3.
Symbol | Description
---|---
< | Start tag open delimiter
element.name | Element name
property | Attribute name
= | Value indicator
" | Literal string delimiter
| Alternative literal string delimiter
value | Value of the attribute
> | Start tag close delimiter

Note that an attribute value must be enclosed in quotation marks. You can use either single quotes (<lie size='big'>) or double quotes (<lie size="massive">), but you cannot mix the two in the same specification.

When you are working without a DTD (none of the XML code shown in today’s chapter requires you to associate a DTD with the XML document), you can simply specify the attribute and its value when you use the element for the first time, as shown in Listing 3.1. When you specify attributes for the same element more than once (as in Lines 3 and 4 of Listing 3.1), the specifications are simply merged.

**LISTING 3.1 SPECIFYING ATTRIBUTES**

```
1: <?xml version="1.0"?>
2: <home.page>
3:  <para number="first">This is the first paragraph.</para>
4:  <para number='second' color="red">This is the second paragraph.</para>
5: </home.page>
```

When the XML processor encounters line 3, it will record the fact that a para element has a number attribute. (Remember that this is in the absence of a DTD, which would explicitly declare what attributes a para element has.) Then, when it encounters line 4, it will record the fact that a para element also has a color attribute.

There is one attribute that is reserved for XML’s own use--the xml:lang attribute. This attribute is reserved to identify the human language in which the element was written. The value of the attribute is one of the ISO 639 country codes; some of the most common language codes are shown in Table 3.4.

**TABLE 3.4 COMMON ISO 639 LANGUAGE CODES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
<td>Arabic</td>
</tr>
<tr>
<td>ch</td>
<td>Chinese</td>
</tr>
<tr>
<td>de</td>
<td>German</td>
</tr>
<tr>
<td>en</td>
<td>English</td>
</tr>
</tbody>
</table>
When there are several versions of a language, such as British and American English, the language code can be followed by a hyphen (-) and one of the ISO 3166 country codes. Some of the most common country codes are shown in Table 3.5. If you have spent much time on the Internet, you may well recognize these as the same codes that are used in email addresses and URLs. An element written in American English could be identified like this (note the cases; the language code is in lowercase and the country code is in uppercase):

<para xml:lang="en-US">My country 'tis of thee.</para>

**TABLE 3.5 COMMON ISO 3166 COUNTRY CODES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Austria</td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
</tr>
<tr>
<td>EN</td>
<td>England</td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
</tr>
<tr>
<td>FR</td>
<td>France</td>
</tr>
<tr>
<td>GR</td>
<td>Greece</td>
</tr>
<tr>
<td>IT</td>
<td>Italy</td>
</tr>
<tr>
<td>JA</td>
<td>Japan</td>
</tr>
<tr>
<td>NL</td>
<td>The Netherlands</td>
</tr>
<tr>
<td>PT</td>
<td>Portugal</td>
</tr>
<tr>
<td>RU</td>
<td>Russia</td>
</tr>
<tr>
<td>US</td>
<td>United States</td>
</tr>
</tbody>
</table>

The codes given in Tables 3.4 and 3.5 are not complete or exhaustive. There is another coding scheme registered by the Internet Assigned Numbers Authority (IANA), which is defined in RFC 1766. And if you really need to, you can devise your own language code. User-defined codes must be prefixed with the string x-, in which case you could declare an element as being in "computer geek" language like this:
Do you grok this code?

Naming Rules

So far you’ve learned about the markup used for elements and attributes, and all the descriptions mention that these markup objects have names. XML has certain specific rules governing which names you can use for its markup objects.

XML’s naming rules are as follows:

- A name consists of at least one letter: a to z, or A to Z.

- If the name consists of more than one character, it may start with an underscore (_) or a colon (:). (Technically, there wouldn’t be anything stopping you having an element called <_, but that would not be very helpful.)

- The initial letter (or underscore) can be followed by one or more letters, digits, hyphens, underscores, full stops, and combining characters, extender characters, and ignorable characters. (These last three classes of characters are taken from the Unicode character set and include some of the special Unicode character symbols and accents. For a full list, refer to the XML recommendation online at http://www.w3.org/XML/REC-xml.)

TIP: The World Wide Web Consortium (W3C) regularly reorganizes its Web site, and the URLs for recommendations, notes, and working drafts change quite often. When you visit their Web site, you will find a pointer to the URL Minder service. This free service is one of the many wonders of the Web. By registering a Web page--any Web page--you will automatically be sent an email message if anything on that page changes. This is an excellent way to keep track of any new developments.

Note that spaces and tabs are not allowed in element names (<one two> would be interpreted as two separate names), and the only punctuation signs allowed are the hyphen (-) and full stop (.)

CAUTION: If you spend any time writing code in any other language (even Java or JavaScript), it’s easy to get into the habit of using an underscore character (_) to separate long names into sensible chunks, as in: This_is_a_Long_Name. This use of underscores is illegal in XML. You would have to rewrite this as This.is.a.Long.Name.

There is no rule that says your choice of a name needs to make sense. As long as you obey the naming rules, you can call XML objects whatever you like and the names can be as long and meaningless as you like. However, one of the major benefits of using XML in the first place is that it is self-describing. If you have elements such as <thingy>, <whatever>, and <huh>, you’re defeating the whole issue. Try to choose names that are at least slightly suggestive of the nature or purpose of the object. Don’t forget that one of the XML’s aims is to be readable by users. Being readable is one thing, but it also helps if they also make sense.

Comments
No self-respecting language, whether it’s a programming language or a markup language, could hold its head up without allowing comments to be added to the code. From a maintenance point of view, it’s also pretty important to have a lasting record of why you did particular things. The best way to document your code is to include the explanation with the code by using comments.

In keeping with the design constraint of keeping XML simple, its comment facilities are also simple. Comments have the form

<!-- this is comment text -->

**CAUTION:** The comment start tag (<!--) and end tag (--> must be used exactly as they are shown here. Inserting spaces or any other characters into these strings can lead to the tags, or anything inside the comment, mistakenly being interpreted by the XML processor as markup.

Provided that you use the comment start tag and end tag correctly, everything in the comment text will be completely ignored by the XML processor. The following comment is therefore quite safe:

<!-- These are the declarations for the <title> and <body> -->

There is only one restriction on what you can place in your comment text: the string -- is not allowed. This keeps XML backward-compatible with SGML. (The string --> will obviously end the comment.)

Comments can be placed anywhere in an XML document outside other markup. The following is therefore allowed:

<para>This is simple <!-- So everyone tells me --> to do.</para>

while this is not allowed:

<para <!-- blatant lie --> >This is simple to do.</para>

**Character References**

Unlike SGML (and, as a result, unlike HTML), which is very much ASCII-based, XML was developed right from the start to support languages other than English. XML therefore has far better support for accented and foreign language characters than either SGML or HTML.

In HTML, you can always enter the code for the foreign language character you want (è would be &egrave;, í would be &iacute;, and û would be &ucirc;). As you will see later in this chapter, these codes are in fact entity references. The abbreviations egrave, iacute, and ucirc are taken from the ISO 8859/1 character set (SGML’s character set), which is derived from the ISO/IEC 646 version of the ASCII alphabet (the first 128 characters). ISO 8859/1 is also the basis for the Microsoft Windows fonts.

Although these character entity references will allow you to deal with most European and Scandinavian languages, things would come to a sudden stop if you tried to display or write in an Asian or Middle Eastern language such as Japanese, Hindi, or Arabic. However, XML is based on
Unicode and the even more extensive ISO/IEC 10646 standards (which even allow the use of Chinese characters). You needn’t concern yourself too much with these character sets now (or not at all if you are only interested in publishing Western languages on the Web), but we will return to this topic later on.

The most important thing you need to know about these exotic characters is that you can still enter them even if your keyboard doesn’t support them. You do this by entering a character reference.

A character reference consists of the string &#, followed by the number of the character in the ISO/IEC 10646 alphabet and terminated by a semicolon (;). The character number may be either a decimal number, in which case you enter the number as-is, or in hexadecimal form, in which case you must precede the number with the letter x, such as x12ABC. For example, the character reference for the copyright symbol (©)—written in HTML as &copy;—is © (in decimal) or © (in hexadecimal).

**Predefined Entities**

Character references allow you to enter characters that you might not be able to enter from your keyboard. A variation on this theme is the set of predefined entities. These are characters that you can enter normally, but you shouldn't because they can easily be mistaken for markup characters. To refresh your memory, the set of predefined entities is shown in Table 3.6.

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp; or &amp;&amp;38;</td>
</tr>
<tr>
<td>‘</td>
<td>' or '</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt; or &gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt; or &amp;&amp;60;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot; or &quot;</td>
</tr>
</tbody>
</table>

You can enter a named entity to represent the character, such as &apos;, or you can enter a character reference, such as &#39;. The character references for the ampersand (&) and the less-than (<) character are special cases, however, so the character references are double-escaped. The reasons for this will be explained in the following section.

**Entity References**

As you remember from yesterday’s discussion of the anatomy of an XML document, entities are normally external objects such as graphics files that are meant to be included in the document. To reference these external entities, you must have a DTD for your XML document. You will learn about these entities when you learn about DTDs, but there is one other type of entity that you can use already, called an internal entity. It can save you a lot of unnecessary typing.

Internal entities look very much like character references, but with one vitally important difference—you must declare an internal entity before you can use it.
Entity Declarations

The declaration of an internal entity has this form:

`<!ENTITY name "replacement text">`

Now, every time the string &name; appears in your XML code, the XML processor will automatically replace it with the replacement text (which can be just as long as you like). Judiciously used, entity references can save you a lot of typing.

The Benefits of Entities

You can almost think of an entity reference as a sort of macro. But whatever you call it, it can be a real time-saver when there is a piece of text that you want to use several times, or even if you want to use some kind of template text.

Consider the example shown in Listing 3.2, in which a copyright statement is used as an entity reference.

LISTING 3.2 USING AN INTERNAL ENTITY

```xml
1:  <?xml version="1.0"?>
2:  <home.page>
3:    <head><title>Title Page</title></head>
4:    <body> <h1>The Title Page</h1>
5:      <para>(c) 1998, &rights;</para>
6:     </body>
7:  </home.page>
```

Given the following declaration for the rights entity:

`<!ENTITY rights "All rights reserved. No part of this book, including interior design...">`

This would result in the following substitution being made in line 5 of Listing 3.2:

`<para>(c) 1998, All rights reserved. No part of this book, including interior design...</para>`

Using an entity reference in this way, you would only have to enter the text once, in the entity declaration, instead of having to search for and change every occurrence of the string in the text. Used in this way, entity references can simplify the task of creating and maintaining XML documents. On Day 8, “XML Objects: Exploiting Entities,” you will learn how to expand this feature to use external entities as a sort of boilerplate text facility, enabling you to declare these text entities in a common document that can be accessed by any number of other documents.

Some of the Dangers of Using Entities

You’ve seen how handy internal entity references can be as a sort of shorthand for entering pieces of text, and as a means of dealing with variable content. Obviously, with a little thought and advance preparation, entity references can save you a lot of time and effort later on.
Naturally, a feature this handy raises a very simple question: "Could I use this to insert markup too?"
It’s an attractive idea and a natural thing to want to do. Can you put markup inside the replacement
text? Well, yes you can... but it’s subject to a few restrictions, and you need to think it out quite
carefully beforehand to avoid some unpleasant surprises.

The first thing you must remember is that XML will process the contents of the entity replacement
text when it expands the entity reference. This means that you must not just escape any markup
characters in the replacement text; you must double escape the characters. Consider this simple
example:

```xml
<!ENTITY dangerous "Black &38; White">

When the XML processor sees the entity reference &dangerous; in the XML document, it will
immediately expand (dereference) the predefined entity before it inserts the replacement text. This
XML code seems harmless enough:

```text
This is not a &dangerous; choice.</text>
```

But let’s look at what happens, step by step:

1. The XML processor sees the entity reference &dangerous; and looks for the replacement
text.
2. Finding Black &#38; White, the XML processor dereferences this to Black & White.
3. The XML processor inserts the replacement text, and the resulting XML code is

```text
This is not a Black & White choice.</text>
```
4. The XML processor then tries to parse the ampersand and reports an error because & has not
been declared as an entity.

### Avoiding the Pitfalls

You’ve seen some of the problems that entity references can create when their contents are
dereferenced. At worst, they can make a complete mess of your XML code. Of course, there’s a way
to avoid these problems--double escape any markup contained in the replacement text, like this:

```xml
<!ENTITY safe "Harry &38;#38; Fred &amp;amp; Joe">

When the XML processor sees the entity reference &safe; in this XML document:

```text
The job was left to &safe; to fix.</text>
```

The expansion will still leave you with valid code. Let’s see what happens as the XML processor
dereferences the entity reference:

1. The XML processor sees the entity reference &safe; and looks for the replacement text.
2. Finding "Harry & Fred; Joe">, the XML processor dereferences this to Harry & Fred; Joe.

3. The XML processor inserts the replacement text, and the resulting XML code is

   <text>The job was left to Harry & Fred & Joe to finish.</text>

4. The XML processor then parses the resulting code, sees the entity reference &name; and dereferences that to produce

   <text>The job was left to Harry & Fred & Joe to finish.</text>

As you can see from these examples, you can escape the markup by using either the entity reference form (in the example, &#38;) or the character reference form (&amp;) of the predefined entity.

Synchronous Structures

Other than these problems, there is one very important restriction on using markup in entities. On Day 2, "Anatomy of an XML Document," you learned that the logical and physical structures in the XML document must be synchronous.

At the time, the restriction might not have made too much sense because it’s difficult to imagine the two structures not being synchronous. Well, here’s an example of the two structures becoming asynchronous: The logical structure is composed of the elements in the XML document and in the replacement text. The physical structure is composed of the document entity (the root entity of the XML document containing the entity reference) and the internal entity (the replacement text). The two objects are discrete physical entities as far as XML is concerned, even though in this case they are actually in the same file. For the two structures to be synchronous, any element that is inside the replacement text must start and finish inside the replacement text (in other words, inside the entity).

The following would be allowed:

```xml
<!ENTITY safe "&#38;60;emph&#62;Harry&#38;60;/emph&#62; and Joe">
<text>The job was left to &safe; to finish.</text>
```

because the dereferenced entity reference would yield this:

```
<text>The job was left to <emph>Harry</emph> and Joe to finish.</text>
```

The following could create a lot of problems, however:

```xml
<!ENTITY unsafe ""&#38;60;emph&62;Harry and Joe">
<text>The job was left to &safe; to finish.</text>
```

even though, when the entity reference has been dereferenced, the resulting markup would actually be quite legal:

```
<text>The job was left to <emph>Harry and Joe</emph> to finish.</text>
```
Although we are still talking about internal entities, which are completely within our control, the restriction is really pretty logical. The same dereferencing mechanism applies for external entities as well as internal entities and, bearing in mind that one of XML’s design goals is to be used easily on the Web, we have absolutely no control over what is contained in external entities. XML’s developers could have made a distinction between internal and external entities, but this would go against two more of XML’s basic design goals--simplicity and clarity.

**Where to Declare Entities**

You have learned what an internal entity reference looks like, and you’ve seen some of the benefits and drawbacks of using entity references. Before we move on to something else, you still need to learn where to put the entity declarations.

Entity references are normally only allowed in the DTD that accompanies the XML document. The declarations of element structures and entities are in fact the only reason for having a DTD at all. You will learn all about DTDs in detail later on; for now, all you need to know is illustrated in Listing 3.3.

**LISTING 3.3 DECLARING AN INTERNAL ENTITY**

```
1:  <?xml version="1.0"?>
2:  <!DOCTYPE home.page [ 
3:    <!ENTITY shortcut "This is the replacement text."> 
4:  ]>
5:  <home.page>
```

Line 1 of Listing 3.3 is the now-familiar XML declaration. Line 2 is a document type declaration. This is the line that will later be used to make the association between the XML document and the DTD that describes its structure.

The document type declaration is the XML statement that declares what type of XML document follows and identifies the document type definition (DTD), which contains the description of the allowed structure of this type of document. (It is quite easy to confuse these two terms.)

The document type declaration takes this form:

```
<!DOCTYPE name external.pointer [ internal.subset ]>
```

where external.pointer points to a separate file that contains the external subset of the DTD. Don’t worry too much about this for now; the trick is that you can leave this out and concentrate on the internal subset of the DTD. The declaration you will need, then, looks like this:

```
<!DOCTYPE name [ internal.subset ]>
```

In this internal subset you can declare as many elements, attributes, and entities as you like without having an external DTD at all.

As you will discover later, there are all sorts of other tricks you can do with the internal DTD subset. Anything you put in the internal subset takes precedence over anything in an external subset. For example, you can declare a default set of global values for a whole suite of XML documents and then
override the global values in an individual XML document when you want to, but that is another story.

Before we leave the subject of DTDs altogether, there is one last thing that you should get into the habit of doing, even if it doesn’t make much sense at this point. Although you aren’t using an external DTD yet, if and when you do, the name that you give to the document type must be the same as the name of the root element in the XML document. This is shown in Listing 3.3, where the document type name (home.page on line 2) is the same as the root (first) element name (line 5). This isn’t a requirement when there isn’t an external DTD, but it is still a good habit to get into.

**CDATA Sections**

You have learned how to escape markup characters by using the predefined entities and character references. Replacing every markup character in a piece of text could be a long and tedious process. Besides, there might be cases when you want to keep all those characters exactly as they are (like when you’re sending the XML code on for further processing by a different application).

The way to do this is to use a CDATA (character data) section, like this:

```xml
<![[CDATA[This is the text < 5 lines > that I want the &!%# XML processor to leave alone!]]>
```

Nothing, absolutely nothing, that appears between the opening tag (<![[CDATA[) and the closing tag (]]) will be recognized as markup. You do not need to escape any markup characters in a CDATA section. (In fact, you can’t anyway because the escape itself won’t be recognized.) The only thing that will be recognized is the end-of-section tag (]]>), so obviously this string cannot be included in a CDATA section. As a logical consequence of this, you cannot put one CDATA section inside another.

**CAUTION:** Using markup characters in a CDATA section like this in an XML document, which is built around markup, rather goes against the grain. An XML processor is intended to prevent you from breaking this unwritten rule, and it’s very unforgiving of any mistakes. The opening string and closing string of a CDATA section must be used exactly as it is shown here. The slightest deviation, a tab or a space character somewhere inside one of the strings, will be punished immediately. The content of the CDATA section will either be treated as markup, or the rest of your document (up to the next CDATA section that is closed properly) will be treated as part of the CDATA section and all the markup will be ignored. You have been warned!

CDATA sections are one of the recommended ways to embed application code (JavaScript, VBasic code, Perl code, and so on) in your XML code. You could place the embedded code in comments, as is often done in HTML documents, but the XML processor is not required to pass the comment text to an application. Therefore, there’s always the risk that the contents of comments will be stripped out before the application sees them.

Even though it is quite legal to declare your own type of element to contain the embedded code (like the <script> element in HTML 4), you’d be implicitly breaking the spirit of generic markup. Nor would it prove to be much help if your embedded code contained characters that could be interpreted as markup, because the contents of these elements would be parsed in the normal way by the XML...
The other way to embed code, and probably the best way, is by using processing instructions.

### Processing Instructions

Probably without even noticing it, you have already seen processing instructions. The XML declaration at the start of every XML document (or at least it should be there) is a processing instruction:

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

XML markup is meant to be generic, and in a perfect world it would be. However, there will always be times when you need to enter instructions for specific applications. One of these applications could be a script interpreter, and so, like CDATA sections, processing instructions are good places to put embedded code. While CDATA sections are purely a way of avoiding characters being interpreted as markup, better still, processing instructions can be targeted to your application. For example, this would allow you to have two or more sets of embedded script code, intended for different processors or interpreters, and identify them separately, as shown in Listing 3.4.

#### LISTING 3.4 EMBEDDING CODE IN PROCESSING INSTRUCTIONS

1:  <para>This is text containing two
2:    processing instructions,<n3:        <?javascript I can put whatever I like here?>
4:        <?perl And I can put whatever I like here too?>
5:    one for each interpreter.</para>

There are no restrictions on the content of the processing instructions (the XML processor doesn’t even consider the content to be part of the document’s character data), but the name that you choose must comply with XML’s naming rules.

### Summary

In this chapter you learned the details of XML’s markup language. You also learned how to declare and use internal entities, as well as some of the benefits and dangers of using them. You were introduced to character references for entering characters not available on your keyboard, and you saw how you can use the characters that are normally reserved for markup in your character data by using character references and the predefined entities.

To conclude, you learned how to use comments and CDATA sections to hide text that could be interpreted as markup by the XML processor, and how you can extend this by using processing instructions to pass code through for processing by other applications.

### Q&A

**Q.** Which of these element names is valid and which is not?

a) `<para 1>`
b) `<para,1>`

c) `<para.1>`

d) `<Pa3A1>`

e) `<para!>`

A. Only c and d are legal; a contains a space, b contains a comma, and e contains an exclamation mark.

Q. What is wrong with the following code fragment?

```xml
<para size="12pt">’twas brillig and
the slithey toves <!-- I’ve no idea
what these are --> did gyre and gymble
in the wabe.</para>
```

A. Comments may not be placed inside elements. They must be outside other markup.

Q. Where do you declare entities?

A. You can declare entities inside either the internal subset or the external subset of the DTD. If you have an external DTD, you will have to create a complete DTD. If you only need the entities and nothing else, you can get away with an internal DTD subset. Entity references in XML documents that have external DTD subsets are only replaced when the document is validated.

Q. Why do I need an XML declaration? It should be obvious that this is XML code.

A. Strictly speaking, you do not need an XML declaration. XML has also been approved as a MIME type, which means that if you add the correct MIME header (xml/text or xml/application), a Web server can explicitly identify the data that follows as being an XML document, regardless of what the document itself says. (MIME, or Multipurpose Internet Mail Extensions, is an Internet standard for the transmission of data of any type via electronic mail. It defines the way messages are formatted and constructed, can indicate the type and nature of the contents of a message, and preserves international character set information. MIME types are used by Web servers to identify the data contained in a response to a retrieval request.)
The XML declaration is not compulsory for practical reasons; SGML and HTML code can often be converted easily into perfect XML code (if it isn’t already). If the XML declaration was compulsory, this wouldn’t be possible.

**Q.** Can I use entities in attribute values as well as in content? This would allow me to parameterize elements.

**A.** Yes and no. You can use entity references in attribute values, but an entity cannot be the attribute value. There are strict rules on where entities can be used and when they are recognized. Sometimes they are only recognized when the XML document is validated. For details, see the XML recommendation itself (http://www.w3.org/XML/REC-xml).

**Q.** Can I put binary data in a CDATA section?

**A.** Technically there’s nothing stopping you, even though it’s really a character data section. Because the XML processor doesn’t consider the contents of a CDATA section to be part of the document’s character data, it will never know or care what you put in there. However, you would have to live with the increase in file size and all the transportation problems that would imply. Ultimately, it would be a shame to jeopardize the portability of your XML documents when there is a far more suitable feature of XML you can use for this purpose. Entities, which you learn about on Day 8, allow you to declare a format and a helper application for processing a binary file (possibly displaying it) and associate it with an XML document by reference.

**Exercises**

1. There are two mistakes in the following fragment of code. What are they?

   ```xml
   <![CDATA [This is the hidden &markup!] ]]>
   ```

   You can check your answers by running the code through one of the XML parsers, as explained on Day 5, "Checking Well-formedness."

2. Yesterday you marked up an email message. Using the appropriate entities, change the markup to turn the XML code into a boilerplate for email messages to anyone.