SYBEX Sample Chapter

XML Complete

Chapter 4: Understanding and Creating Entities

Copyright © 2001 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4033-5

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

SYBEX Inc.
1151 Marina Village Pkwy.
Alameda, CA 94501
USA
Phone: 510-523-8233
www.sybex.com

Chapter 4

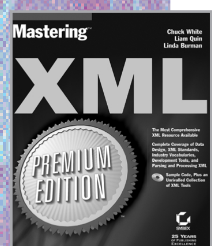
Understanding and Creating Entities

If you've developed HTML-based Web pages, you probably have used entities quite often, even if you've done most of your development with WYSIWYG editors. If you use the < symbol in a WYSIWYG editor and look at the source code, you'll find that the actual HTML code used to represent the < looks like this:

```
&lt;
```

This is an example of an entity. The < symbol needs to be represented as an entity, because the parser would confuse < symbols with < tags. For example, if an HTML document contains HTML code examples, entities provide a simple way to represent HTML tags without confusing the parser.

According to the XML specification, XML documents consist of a set of storage units. These storage units are called *entities.*



Adapted from *Mastering XML™, Premium Edition*, by Chuck White, Liam Quin, and Linda Burman

ISBN 0-7821-2847-5 1,155 pages $49.99

XML uses a few of the same entities as HTML to represent markup that should not be parsed by XML syntax rules. However, XML considerably extends the power of entities because you can define them just as easily as you can define elements. This means that entities can consist of a broad range of objects. These objects can include just about any programming-based object that comes to mind, including binary graphics, word processing files, or multimedia applets. This chapter will look at entities and guide you through the process of understanding and creating them. Specifically, we'll examine the following:

- What entities are
- When to use entities
- The different types of entities available to XML authors
- How to use internal and external entities
- How to use parameter and general entities
- How to be sure your entity markup is legal
- The use of system and public identifiers
- Ways of harnessing the power of entities by learning from examples that demonstrate how entities are defined in a document type definition (DTD) for referencing in a document instance

Learning the Basics about Entities

The *document entity* is the most important entity in an XML document and is actually one of only two kinds of entities that are allowed to exist without having a name assigned to them (the other kind of unnamed entity is the external DTD subset). This entity is the first thing the XML processor encounters when parsing a document. It is also referred to as the document root, and it provides programmatic access to the rest of the document. The reason the document entity is important is that, at the end of the day, it's the only thing the XML specification requires an XML parser to read.

WARNING

This should not be mistaken for the root *element* of a document, which is the first element in an XML document and contains any other elements that exist.

This chapter is more concerned with two specific types of entities: general entities and parameter entities. The HTML entities described in this chapter's opening paragraphs are a type of general entity called predefined entities. Entities can be used as a kind of shorthand that allows you to embed blocks of text or even entire documents and files into an XML document. This makes updating documents across networks very easy. Entities also allow you to represent special characters like markup. You can even use entities in a DTD to cut down on the amount of code.

Within the scope of general entities and parameter entities are four other types of entities, which can be considered subsets of general and parameter entities:

Internal entities These are entity references that refer to entities whose definitions can be found entirely *within* a document's DTD.

External entities These are entity references that refer to entities whose definitions can be found outside of a document.

Parsed entities These are entities that the XML processor can and will parse.

Unparsed entities These are entities that are not parsed by the XML processor, but instead are handed off to another application for processing and are often described by binary mechanisms, such as those in image files.

Using Entities

Have you ever run a mail-merge function in a word processing program? In a mail merge, you develop a database of names and addresses and bind them to a word processing document with some markup. The markup tells the word processor where in the word processing document the address information from the database should go. If you've used mail merges, you've seen the concept of entities at work. Instead of character data, such as an address block, XML allows a wide variety of data to be used as an entity.

Entities operate on a similar principle to mail-merge functions in the sense that an entity acts as a replacement mechanism. That's why entities are such great shorthand for XML documents. Some of the uses for entities include the following:

- Denoting special markup, such as the > and < tags.
- Managing binary files and other data not native to XML.
- Reducing the code in a DTD by bundling declarations into entities.
- Offering richer multilanguage support.
- Repeating frequently used names in a way that guarantees consistency in spelling and use.
- Providing for easier updates. By using entities in your markup for items you know will be changed later—such as sports scores or software version changes—you greatly improve dynamic document automation.
- Managing multiple file links and interaction.

Making Sure Your Entity Markup Is Legal

Entity syntax rules vary depending on the kind of entity you are using, but like everything else in XML, they're pretty straightforward. When you use an entity within an XML document, you must follow five rules:

- The entity must be declared in the DTD. If you're creating an XML document that is not being validated against a DTD or schema, you need to create enough of a DTD yourself within the XML document to at least declare the entity that you are using. The exceptions to this rule are the predefined entities of XML, but there are only five of them, so they're easy to remember (you'll visit these a bit later), although many people recommend you declare these also.
- A general entity referenced within an XML document must be surrounded by the ampersand (&) on one end and the semicolon (`;`) at the other (`&myEntity;`).
- The name of an entity must begin with a letter or underscore (_) but can contain letters, underscores, whole numbers, colons, periods, and/or hyphens.

- An entity declaration cannot consist of markup that begins in the entity declaration and ends outside of it.
- A parameter entity must be declared with a preceding percent sign (%) with a white space before and after the percent sign, and it must be referenced by a percent sign with no trailing white space. A typical parameter entity declaration looks like this: `<!ENTITY % myParameterEntity "myElement">`.

Understanding General Entities

The jargon and semantics involving XML can be overwhelming, and one example of this is entities. There are external general parsed entities and external general unparsed entities, as well as parameter entities and internal general entities. This all can get pretty confusing. Our advice is to keep it simple and focus on the two kinds of entities, general entities and parameter entities, that have a clear difference in usage. The underlying concepts are the same in that they both act as a kind of shortcut.

General entities are easier to describe by what they are not than what they are: If it's not a parameter entity, then it's a general entity. Parameter entities can appear only in a DTD. General entities appear in the main XML document (called the *document instance*) that begins with the root element. It's actually more accurate to say that the entity *reference* appears in the document instance, whereas the entity definition is found in the DTD. In fact, you can't create an entity reference within a document instance without validating it against a declaration made in either an internal or external DTD or schema.

We'll explore general entities by reviewing four different kinds:

Predefined entities These are entities defined by the XML specification for managing what might normally be considered markup so that you can represent "markup" symbols without the parser mistaking them for actual XML markup.

Character references These are character sets that represent specific characters in a given character set, which makes it possible, for example, to add special symbols using a specific markup.

Entity references These refer to a declaration made within a document and can be parsed by the XML processor.

Unparsed entities These are entities that are not parsed by the XML processor and, instead, need to be passed off to another application.

Using Predefined Entities

XML has several predefined, or built-in, entities that can be invoked without any special declarations. The simplest kinds of general entities, they are the only kinds that require no declarations within a DTD or schema. These predefined entities are listed in Table 4.1.

TABLE 4.1: Predefined XML Entities

Entity Name	Character Reference	Corresponding Character
<	<	<
>	>	>
'	'	'
"e;	"	"
&	&	&

The predefined entities in Table 4.1 are not really so much shortcuts as they are a means for preventing the XML processor from throwing an error or refusing to parse your XML document. If you look at the symbols in Table 4.1, you can see that they are symbols that might often be encountered in XML markup, and thus act as a means for representing these characters when you don't want them interpreted as markup. Using an entity from Table 4.1, you could write the following line of code, without worrying about DTD validation. See Figure 4.1 to see how this is rendered in Internet Explorer 5 (IE 5).

```
<MYELEMENT>1 &lt; 2.</MYELEMENT>
```

These are the easiest kinds of entities to use. But they don't offer much beyond a few very simple, specific tasks. XML provides a method for declaring entities in a DTD for reference later in your document instance.

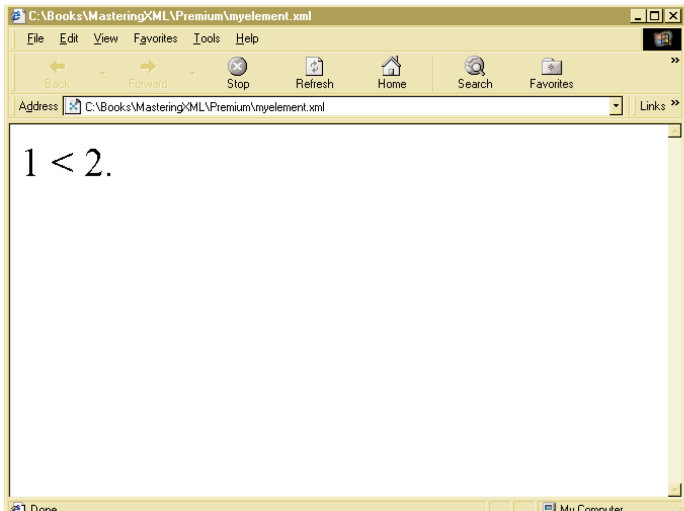


FIGURE 4.1: A predefined entity rendered in IE 5

Generally, you are best off using the entity names in an XML document. If you are seeking compatibility with legacy SGML documents, you should declare them using the character reference in the document's DTD. This is an example of a typical predefined entity declaration in a DTD:

```
<!ENTITY lt "&;#60;>
```

XML processors, however, are required by the XML specification to recognize these entities whether they are declared or not. It's when you're working with SGML applications that declaring them makes sense.

Working with Character References

You can include special kinds of references within your XML documents with *character references*. Character references are similar in look to entity references, but they refer to specific characters (such as accented letters) using a special numbering system called Unicode and don't need to be declared. Unicode is an encoding system that maps character data across the world's language boundaries. You can use a character reference in your XML document as long as you know the corresponding Unicode reference, but some developers might choose to create an entity reference to it in your DTD so that others know what you're trying to do.

The default character set for XML is the ISO-Latin-1 character set, which is what most English-speaking and Western European developers

will use. There are numerous other character sets that can be included, but they need to be declared in the DTD. If you use ISO-Latin-1, you don't need to declare either the character set or the character references from that set. However, there are many languages of the world, and attempts are being made to include the capacity to manage these languages.

NOTE

The XML 1 specification requires XML processors to support character references mapped to the ISO/IEC 10646 character set, which is a Unicode encoding. To gain access to the world of Unicode and the character sets available for non–Western European languages, visit `charts.unicode.org/Unicode.charts/normal/Unicode.html`**. This Web site provides an extensive series of charts for a variety of languages, and the corresponding hexadecimal code to include in entity and declarations and references. For the most recent Unicode specification, visit** `www.unicode.org/unicode/reports/tr8.html`**.**

The ISO-Latin-1 character set includes most of the ASCII set of characters. ASCII is an acronym for American Standard Code for Information Interchange and is the basis for most computer character sets prior to the introduction of Unicode. When you choose the Save As Text option in an application, the vast majority of English-based applications create an ASCII file rather than a binary file. Even programs that save text files in Unicode will usually make that choice clear to the user. An ASCII file is saved character by character, which means that the decimal number 83 represents the letter S. To refer to this specific number in XML, you need to add the & and `;` characters. This means the full XML character reference is `&83;`. Unfortunately, despite the rather authoritative-sounding nature of the ASCII name, it is not a rigid standard, and it is subject to some unpleasant vagaries. Still, it's a key part of understanding how text-encoding functions work.

You use a character reference using ASCII or IS0-Latin-1 (which includes non-control ASCII characters) in XML by looking in Table 4.2 for the character you want to include in the XML file and then wrapping it in the ampersand (&) and semicolon (`;`).

TABLE 4.2: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
0	Null character	111	o	
1	Start of heading	112	p	
2	Start of text	113	q	
3	End of text	114	r	
4	End of transmission	115	s	
5	Enquiry	116	t	
6	Acknowledge	117	u	
7	Terminal bell	118	v	
8	Backpace (non-destructive)	119	w	
9	Horizontal tab (move to next tab position)	120	x	
10	Line feed	121	y	
11	Vertical tab	122	z	
12	Form feed	123	{	
13	Carriage return	124	|	
14	Shift out	125	}	
15	Shift in	126	~	Tilde
		160		Non-breaking space
16	Data link escape	161	¡	Inverted exclamation
17	Device Control 1, normally XON	162	¢	Cent sign
18	Device Control 2	163	£	Pound sterling
19	Device Control 3, normally XOFF	164	¤	General currency sign
20	Device Control 4	165	¥	Yen sign
21	Negative acknowledge	166	¦	Broken vertical bar
22	Synchronous idle	167	§	Section sign
23	End transmission block	168	¨	Umlaut (dieresis)
24	Cancel line	169	©	Copyright

TABLE 4.2 continued: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
25	End of medium	170	ª	Feminine ordinal
26	Substitute	171	«	Left angle quote, opening guillemet
27	Escape	172	¬	
28	File separator	173		Soft hyphen
29	Group separator	174	®	Registered trademark
30	Record separator	175	¯	Macron accent
31	Unit separator	176	°	Degree sign
32	Space	177	±	Plus or minus
33	!	178	2	Superscript two
34	"	179	3	Superscript three
35	#	180	´	Acute accent
36	$	181	µ	Micro sign
37	%	182	¶	Paragraph sign
38	&	183	·	Middle dot
39	'	184	¸	Cedilla
40	(	185	1	Superscript one
41	)	186	º	Masculine ordinal
42	*	187	>>	Right angle quote, closing guillemet
43	+	188	¼	One-fourth fraction
44	,	189	½	One-half fraction
45	-	190	¾	Three-fourths fraction
46	.	191	¿	Inverted question mark
47	/	192	À	Capital *A*, grave accent ("À")
48	0	193	Á	Capital *A*, acute accent ("Á")
49	1	194	Â	Capital *A*, circumflex accent ("Â")

TABLE 4.2 continued: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
50	2	195	Ã	Capital *A*, tilde ("Ã")
51	3	196	Ä	Capital *A*, dieresis or umlaut mark ("Ä")
52	4	197	Å	Capital *A*, ring ("Å")
53	5	198	Æ	Capital *AE*, diphthong (ligature) ("Æ")
54	6	199	Ç	Capital *C*, cedilla ("Ç")
55	7	200	È	Capital *E*, grave accent ("È")
56	8	201	É	Capital *E*, acute accent ("É")
57	9	202	Ê	Capital *E*, circumflex accent ("Ê")
58	:	203	Ë	Capital *E*, dieresis or umlaut mark ("Ë")
59	;	204	Ì	Capital *I*, grave accent ("Ì")
60	<	205	Í	Capital *I*, acute accent ("Í")
61	=	206	Î	Capital *I*, circumflex accent ("Î")
62	>	207	Ï	Capital *I*, dieresis or umlaut mark ("Ï")
63	?	208	D	Capital *Eth*, Icelandic ("Ð")
64	@	209	Ñ	Capital *N*, tilde ("Ñ")
65	A	210	Ò	Capital *O*, grave accent ("Ò")
66	B	211	Ó	Capital *O*, acute accent ("Ó")

TABLE 4.2 continued: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
67	C	212	Ô	Capital *O*, circumflex accent ("Ô")
68	D	213	Õ	Capital *O*, tilde ("Õ")
69	E	214	Ö	Capital *O*, dieresis or umlaut mark ("Ö")
70	F	215	x	Multiply sign
71	G	216	Ø	Capital *O*, slash ("Ø")
72	H	217	Ù	Capital *U*, grave accent ("Ù")
73	I	218	Ú	Capital *U*, acute accent ("Ú")
74	J	219	Û	Capital *U*, circumflex accent ("Û")
75	K	220	Ü	Capital *U*, dieresis or umlaut mark ("Ü")
76	L	221	Ý	Capital *Y*, acute accent ("Ý")
77	M	222	Þ	Capital *THORN*, Icelandic ("Þ")
78	N	223	ß	Small sharp *s*, German (sz ligature) ("ß")
79	O	224	à	Small *a*, grave accent ("à")
80	P	225	á	Small *a*, acute accent ("á")
81	Q	226	â	Small *a*, circumflex accent ("â")
82	R	227	ã	Small *a*, tilde ("ã")
83	S	228	ä	Small *a*, dieresis or umlaut mark ("ä")
84	T	229	å	Small *a*, ring ("å")

TABLE 4.2 continued: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
85	U	230	æ	Small *ae*, diphthong (ligature) ("æ")
86	V	231	ç	Small *c*, cedilla ("ç")
87	W	232	è	Small *e*, grave accent ("è")
88	X	233	é	Small *e*, acute accent ("é")
89	Y	234	ê	Small *e*, circumflex accent ("ê")
90	Z	235	ë	Small *e*, dieresis or umlaut mark ("ë")
91	[	236	ì	Small *i*, grave accent ("ì")
92	\	237	í	Small *i*, acute accent ("í")
93	]	238	î	Small *i*, circumflex accent ("î")
94	^	239	ï	Small *i*, dieresis or umlaut mark ("ï")
95	_	240	ð	Small *eth*, Icelandic ("ð")
96	`	241	ñ	Small *n*, tilde ("ñ")
97	a	242	ò	Small *o*, grave accent ("ò")
98	b	243	ó	Small *o*, acute accent ("ó")
99	c	244	ô	Small *o*, circumflex accent ("ô")
100	d	245	õ	Small *o*, tilde ("õ")
101	e	246	ö	Small *o*, dieresis or umlaut mark ("ö")
102	f	247	÷	Division sign
103	g	248	ø	Small *o*, slash ("ø")

TABLE 4.2 continued: ASCII and ISO-Latin-1 Character Sets

Character Reference	Character Explanation	Character Reference	Character	Character Explanation
104	h	249	ù	Small *u*, grave accent ("ù")
105	i	250	ú	Small *u*, acute accent ("ú")
106	j	251	û	Small *u*, circumflex accent ("û")
107	k	252	ü	Small *u*, dieresis or umlaut mark ("ü")
108	l	253	ý	Small *y*, acute accent ("ý")
109	m	254	þ	Small *thorn*, Icelandic ("þ")
110	n	255	ÿ	Small *y*, dieresis or umlaut mark ("ÿ")

Let's look at a specific example of a character reference to see how handy they can be. In Listings 4.1–4.6, we are including characters from the ISO-Latin-1 character set. There are three groups of listings appearing in pairs. The first listing of each pair is an XSLT style sheet that renders an XML document (you could use any XML document as its source because it's a generic style sheet). In each case, we've bolded the character reference to show first the character reference, then the replacement text as it would be rendered if you were using XSLT to transform a document into HTML. You can learn more about the inner workings of XSLT in Chapter 12, "Transforming XML: Introducing XSLT."

Listing 4.1: Generating an @ Sign with a Character Reference Using XSLT

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output indent="yes"/>
   <xsl:template match="/">
      <html>
         <head>
            <title>Contact Info</title>
```

```
          </head>
        <body
         style="font-size:12px;
                font-family: Verdana, Arial, Helvetica;">
         contact me at chuckw&#64;javertising.com
        </body>
      </html>
    </xsl:template>
</xsl:stylesheet>
```

Listing 4.2: The Result of a Tree Generated by XSLT after Interpreting a Character Reference

```
<html>
   <head>
      <META http-equiv="Content-Type"
            content="text/html; charset=UTF-16">
      <title>Contact Info</title></head>
   <body
    style="font-size:12px;
           font-family: Verdana, Arial, Helvetica;">
    contact me at chuckw@javertising.com
   </body>
</html>
```

Listing 4.3: Generating an Æ-Combined Character with a Character Reference Using XSLT

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
   <xsl:output indent="yes"/>
   <xsl:template match="/">
      <html>
         <head>
            <title>Contact Info</title>
         </head>
         <body
          style="font-size:12px;
                 font-family: Verdana, Arial,
                 Helvetica;">
          I have an ancestor who was the
          king of a small English province.
          His name was &#198;thelred.
```

```
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

Listing 4.4: The Result of a Tree Generated by XSLT after Interpreting a Character Reference

```
<html>
   <head>
      <META http-equiv="Content-Type"
       content="text/html; charset=UTF-16">
      <title>Contact Info</title>
   </head>
   <body style="font-size:12px;
         font-family: Verdana, Arial, Helvetica;">
    I have an ancestor who was the king of a
    small English province. His name was Æthelred.
   </body>
</html>
```

Listing 4.5: Generating a © Sign with a Character Reference Using XSLT

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
   <xsl:output indent="yes"/>
   <xsl:template match="/">
      <html>
         <head>
            <title>Contact Info</title></head>
         <body style="font-size:12px;
               font-family: Verdana, Arial,
               Helvetica;">
          &#169;2001 Sybex, Inc.
         </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

Listing 4.6: The Result of a Tree Generated by XSLT after Interpreting a Character Reference

```
<html>
```

```
    <head>
       <META http-equiv="Content-Type"
             content="text/html; charset=UTF-16">
       <title>Contact Info</title>
    </head>
    <body style="font-size:12px;
                 font-family: Verdana, Arial,
                 Helvetica;">
     ©2001 Sybex, Inc.
    </body>
</html>
```

In these listings, each character reference was replaced with a specific character. As tools mature, you can expect to find tables similar to Table 4.1 built into XML editing applications.

Using Parsed Entities

Another kind of entity reference is those that refer to a declaration made within a document and that can be parsed by the XML processor. These are called *parsed entities*, which some people refer to as *text entities* because their definitions are represented as a string of text within quotation marks in the validating DTD. For example, consider the following lines of code:

```
<!ENTITY ADTEXT "This content belongs to the INCOLUMNCONTENT
element. The INCOLUMNCONTENT element is nested within the
INCOLUMN element, as is another child of the INCOLUMN
element, the INCOLUMNSIZE element. One example, without style
sheets, shows how it looks in Internet Explorer. The example
without style sheets uses an entity to place this text from
an outside source. The other example uses style sheets, and
the text is typed in, and an entity is not used.">
```

The preceding line of code may look familiar from the previous chapter on attributes, where you learned that one of the valid attribute types was ENTITY. The preceding lines of code are an example of an internal, parsed entity declaration. Such a lengthy declaration is also a good example of why external entity references are convenient (but not as well supported). The XML processor will parse this entity when it is referred to in the XML document instance:

```
<MYELEMENT>&ADTEXT;</MYELEMENT>
```

When the XML document is processed, the entity &ADTEXT; is replaced by the definition that is contained between the quotation marks

(see Figure 4.2). If an entity can be used in such a way, it is considered a parsed entity. Either an entity is parsed or it is not parsed. The processor will pass on an unparsed entity to another application for processing.

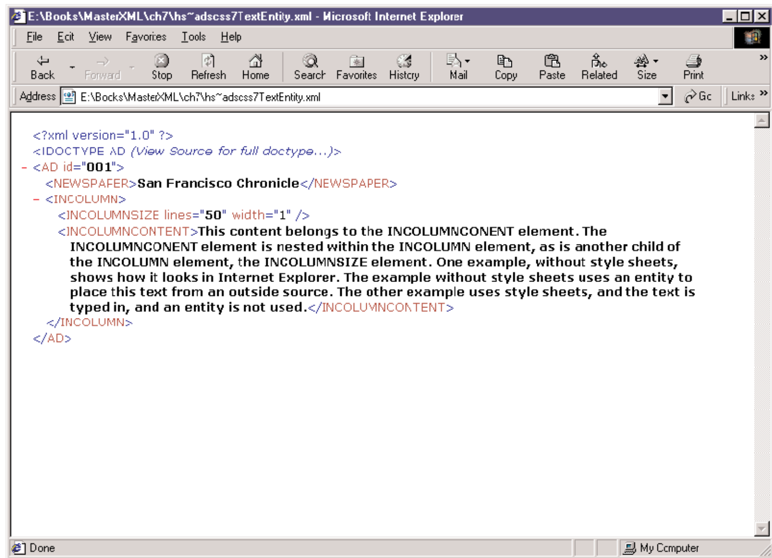


FIGURE 4.2: The entity reference is replaced by the actual defining text.

Imagine, now, that the text in the entity declaration changes for some reason. Perhaps a new version of software has come out and the descriptions change. Or, in our case, some general silliness ensues. Thus the preceding example becomes:

```
<!ENTITY ADTEXT "This content has been completely changed in
accordance to Best Practices, because it was best to do so.">
```

Figure 4.3 shows the result of this change. These kinds of changes become even more significant when external DTDs are used instead of internal DTDs, because global changes could be made to multiple documents: one edit, many changes.

NOTE

Note that Figures 4.2 and 4.3 shows a tree representation of the XML document, which is how IE 5 renders XML documents not attached to style sheets. Technically, however, IE 5 uses a default XSL style sheet to render the tree if a style sheet is not named for the XML document that is parsed by IE 5's XML processor.

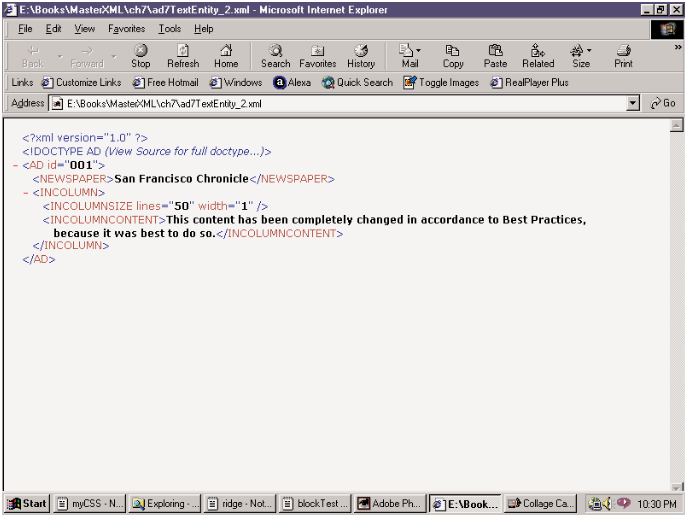


FIGURE 4.3 The XML processor interprets changes in entity definitions.

Often, an entity or character reference will be used in an ENTITY declaration. Listing 4.7 declares a copyright entity named rights by using the IS0-Latin decimal number that maps to that letter.

Listing 4.7: Using a Character Reference to Declare an Entity

```
<?xml version="1.0"?>
<!DOCTYPE fragment [
<!ELEMENT fragment (#PCDATA)>
<!ENTITY rights "&#169;">
]>
<fragment>&rights;2001 Chuck White</fragment>
```

Using the XSLT document in Listing 4.8, you can transform the document into a text document and get the results you would expect. To learn how to create XSLT documents, refer to Chapter 12.

Listing 4.8: Using XSLT to Output a Parsed Entity

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output method="text"/>
   <xsl:template match="/">
   <xsl:apply-templates /></xsl:template>
```

```
    <xsl:template match="fragment">
    <xsl:apply-templates/></xsl:template>
  </xsl:stylesheet>
```

The results of applying the XSLT transform look like this:

```
©2001 Chuck White
```

Managing Unparsed Entities

Whereas parsed entities can be referenced within an element, an *unparsed entity* must appear as an attribute value. An unparsed entity, as its name implies, is not parsed by the XML processor. Processors need to know what to do with these entities, so the instructions must be laid out in the DTD. These instructions appear in the form of an ENTITY declaration that includes the file type of the entity the processor can expect. In addition, the DTD is required to include a NOTATION declaration that indicates to the processor what software will be handling the request.

The interesting thing about such NOTATION declarations is that the processor doesn't really care too much about it. If an invalid path is declared as part of the notation, the XML parser simply moves on to the next order of business. Consider the following line of code:

```
<!ENTITY ADGRAPHIC SYSTEM "border.png" NDATA PNG>
<!NOTATION PNG SYSTEM "MYGRAPHICSPROGRAM.EXE">
```

Without getting too bogged down by the intricacies of DTD authoring (which are covered thoroughly in Chapter 6, "Creating Your Own DTD"), think about the ENTITY declaration in the preceding lines of code as being part of a larger DTD. The ENTITY declaration is declared using, first, the ENTITY keyword, then the name of the entity (ADGRAPHIC in the preceding example). NDATA is an identifier and PNG is the file type. The NDATA keyword and a value indicating the file type (PNG in this case) are required.

Next, you see the NOTATION declaration. This declaration simply tells the XML processor where it can find a program to which it can pass along the information. The XML processor can choose to process the information itself if it wants to, but the DTD author needs to give the XML processor the choice. In other words, the DTD author must include the name (and path) of the software that should process the binary. Figure 4.4 shows what happens when the XML processor in IE 5 doesn't encounter the NOTATION element in an XML document's DTD.

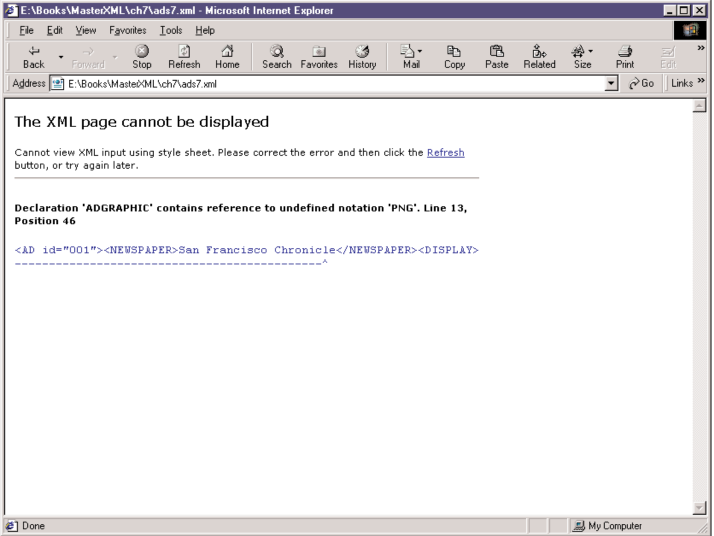


FIGURE 4.4 IE 5 will protest if you don't include a NOTATION declaration for an unparsed entity.

There is no requirement whatsoever that an XML processor actually process an unparsed entity, but it can if it wants. This is true even if that unparsed entity is an external XML document. The main thing to remember about unparsed entities, because they are often binaries of some kind (like images, music, and even Excel or Word documents), is that you're not really looking for a way to render them within an XML browser. Really, there is no such thing as an XML browser. XML, after all, is not a presentation markup language. An XML processor doesn't care what you do with the image or multimedia entity described in your XML document.

The processor is concerned with helping to manage your entities, making them searchable and easy to update, adding structure to the whole process of document creation, and passing along the entity reference to a processing application that can handle it. This doesn't mean that no errors will be generated when the entity is processed, but if you encounter an error message and are certain it's related to an unparsed entity, the message is likely being generated by the application that is handling the entity, not the XML processor. This is an important consideration when diagnosing problems associated with XML documents and their entities.

An unparsed entity is not referenced in the document instance the same way as other entity references—with a beginning & character and ending ; character. In the earlier example, in order to reference the ADGRAPHIC entity, you need to include it as an attribute value. Unparsed entities aren't, and can't be, referenced in elements. They must appear in the document instance as attribute values, and in order to be referenced as an attribute value, the attribute that references the unparsed entity must be declared in the DTD as able to work with entity types. This means that when you're looking at a DTD, it needs to have one more line of code than what you encountered a bit earlier. So add that line of code to your DTD as follows:

```
<!ENTITY ADGRAPHIC SYSTEM "border.png" NDATA PNG>
<!NOTATION PNG SYSTEM "MYGRAPHICSPROGRAM.EXE">
<!ATTLIST DISPLAYCONTENT src ENTITY #REQUIRED>
```

Generally, such an attribute declaration will be made immediately after the element declaration to which it belongs, although there is no requirement for this (though this has become a convention that most XML developers follow). For the time being, let's assume that an element declaration was made for an element named DISPLAYCONTENT a bit earlier in the DTD. The attribute list declaration for the DISPLAYCONTENT element states that there is an src attribute and that the attribute type is an entity. Now all you need to do is name the entity as the value of the attribute in the DISPLAYCONTENT element's src attribute as follows:

```
<DISPLAYCONTENT src="ADGRAPHIC"/></DISPLAY>
```

Notice that when you include the entity as an attribute value, there is no preceding & and no ending ; character surrounding the entity. You're not restricted to using binary graphics as the entity file format. You can use any number of file formats. The only limitation is the ability of the XML processor to pass it on to an application that can do something with the entity.

Using Internal and External Entities

Not only is every entity either parsed or unparsed, but every entity is also either internal or external. These are not mutually exclusive concepts. For instance, you can have an external parsed entity or an external unparsed entity.

An internal entity is one that is defined locally within a DTD, such as the earlier example, which is repeated as follows:

```
<!ENTITY ADTEXT SYSTEM "This content belongs to the INCOLUMN-
CONTENT element. The INCOLUMNCONTENT element is nested within
the INCOLUMN element, as is another child of the INCOLUMN
element, the INCOLUMNSIZE element. One example, without style
sheets, shows how it looks in Internet Explorer. The example
without style sheets uses an entity to place this text from
an outside source. The other example uses style sheets, and
the text is typed in, and an entity is not used.">
```

Generally, you want to avoid writing long definitions, like the previous, in your DTD. You also may want to develop a system by which it's easier to manage a large group of entities. One way to do this is with external entities. For instance, in the preceding example, you could include a string of text within the quotation marks in a separate text file, call it `adText.txt`, and then refer to that file in a Uniform Resource Identifier (URI) in the DTD entity declaration as follows:

```
<!ENTITY ADTEXT SYSTEM "adText.txt">
```

The preceding line of code accomplishes the same thing as the lines of code before it, but makes for a more compact DTD. The end result in the document instance, however, will be the same. When the entity reference is made in the document instance, the actual rendering of the document displays the replacement text contained in the file `adText.txt`. The result is the same screen that you saw in Figure 4.2.

Let's examine the line of code a bit more closely.

```
<!ENTITY ADTEXT SYSTEM "adText.txt">
```

The code, which is an entity declaration within a DTD, begins with the keyword `ENTITY`. The next word, `ADTEXT`, is the name of the entity. The keyword `SYSTEM` is called an *external identifier*, which can take the value of either `SYSTEM` (*system identifier*) or `PUBLIC` (*public identifier*). The external identifier `SYSTEM` refers to a URI. In the preceding example, `"adText.txt"` is a relative URI and could just as easily been an absolute URI such as `"http://www.myDomain.com/adText.txt"`.

Generally you'll see the public identifier for entities either with built-in processor support or with some other kind of special retrieval mechanism. They're designed for intranet or extranet use—or any kind of a situation where the entity is common knowledge among the systems accessing its use. (You'll learn more about public identifiers and system identifiers in Chapter 6.)

You've already seen an unparsed external entity when the use of the ADGRAPHIC entity was demonstrated a bit earlier in the chapter. You can't have an unparsed internal entity because XML rules forbid it. To include an unparsed internal entity, you'd have to include the actual code in the definition that lies between the quote marks in the entity declaration, which isn't particularly realistic.

To review the syntactical difference between internal and external entities, keep the following in mind:

- Internal entities are declared without any external identifiers (the SYSTEM or PUBLIC keywords) in the DTD.
- External entities must be declared with an external identifier (the SYSTEM or PUBLIC keywords) in the DTD.
- External unparsed entities must have the NDATA keyword included in their DTD declarations.
- External unparsed entities must have a NOTATION declaration indicating to the XML processor what application should handle the entity.

Harnessing the Power of Entities

Now that you've had a chance to look at entities from a conceptual standpoint, it's time to see how they work in the guts of an XML document. If you're uncomfortable with all the DTD jargon, you may want to read Chapter 6, which will provide a good grasp of DTD development. Unfortunately, in the meantime it's not really possible to discuss entities without at least taking a glimpse into their DTD syntax.

We'll focus our practical examination of general entities by looking more closely at two kinds of general entities—parsed and unparsed—and provide an example of how to recognize them in a DTD and then apply them to an XML document instance.

Developing General Entities

We'll start by taking a look at an XML document with its DTD included (this is called an *internal DTD*) to make it easier to see how entities are referenced. Take a look at Listing 4.9 and see if you can track down the

entities in the listing's internal DTD. The DTD starts with the [character and ends with the] character. What follows that is the document instance, which is the main part of the XML document that you would have if you split Listing 4.9 into two listings, one with the DTD and one with the document instance.

Listing 4.9: Incorporating Entities into an XML Document Using an Internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE AD [
<!ELEMENT AD (NEWSPAPER, (DISPLAY*, INCOLUMN))>
<!ENTITY ADTEXT_1 "Check out this entity:">
<!ENTITY ADTEXT_2 SYSTEM "adText.txt">
<!ATTLIST AD id ID #REQUIRED>
<!ELEMENT NEWSPAPER (#PCDATA)>
<!ELEMENT INCOLUMN (INCOLUMNSIZE, INCOLUMNCONTENT)>
<!ELEMENT INCOLUMNSIZE (#PCDATA)>
<!ELEMENT INCOLUMNCONTENT (#PCDATA)>
<!ATTLIST INCOLUMNSIZE lines NMTOKEN #REQUIRED>
<!ATTLIST INCOLUMNSIZE width (1 | 2 | 3 | 4
 | 5 | 6 | 7) #REQUIRED>
<!ELEMENT DISPLAY (DISPLAYSIZE, DISPLAYCONTENT)>
<!ELEMENT DISPLAYSIZE (#PCDATA)>
<!ELEMENT DISPLAYCONTENT (#PCDATA)>
<!NOTATION PNG SYSTEM
"D:\Program Files\Photoshop 4.0 LE\Photosle.exe">
<!– The processor does not actually
care where the program that executes the binary
lies, although the binary won't
run if the path isn't set correctly –>
<!ATTLIST DISPLAYCONTENT src ENTITY #REQUIRED>
<!ENTITY ADGRAPHIC SYSTEM "border.png" NDATA PNG>
<!ATTLIST DISPLAYSIZE lines NMTOKEN #REQUIRED>
<!ATTLIST DISPLAYSIZE width (1 | 2 | 3 | 4
| 5 | 6 | 7) #REQUIRED>
]>
<AD id="a001">
   <NEWSPAPER>San Francisco Chronicle</NEWSPAPER>
   <DISPLAY>
      <DISPLAYSIZE lines="636" width="2"/>

      <DISPLAYCONTENT src="ADGRAPHIC"/>
```

```
    </DISPLAY>
    <INCOLUMN>
       <INCOLUMNSIZE lines="2" width = "2"/>
       <INCOLUMNCONTENT>&ADTEXT_1; &ADTEXT_2;
       </INCOLUMNCONTENT>
    </INCOLUMN>
</AD>
```

Did you find the entity declaration ADTEXT_1 in the DTD in Listing 4.9? You can see that a string of text defines the ADTEXT_1 entity. Note that no external system or public identifier was used. The reason for this is that if you use an external identifier, the processor will attempt to find an external entity. But the first entity is an internal parsed entity consisting only of a string. So you leave out the identifier and simply define the entity with the string between quotes. If you use an external identifier with a parsed entity, the XML processor will return an error like this when you load the XML document:

```
Error while parsing entity 'ADTEXT_1'. Could not load 'Check
out this entity:'. The system cannot locate the resource
specified. Line 25, Position 1
```

The reason for this error is that the system or public identifier tells the processor to look for that string of text as a URI.

Now take another look at Listing 4.9 and see if you can find the next entity. This one is called ADTEXT_2. Notice how this entity is defined with the system identifier, which tells you you're dealing with an external entity. Now it makes sense for the XML processor to look for something external, in this case a text file named `adText.txt`.

Next, you should find an external entity called ADGRAPHIC, which is defined by a relative URI. The graphic is called `border.png`. The `.png` extension refers to the Portable Network Graphics (PNG) format, which is a graphic format somewhat akin to a high-powered GIF file (lots of color and transparency options). So the processor will look for the graphic and pass it on to another application for processing.

Now that you've located the three entity declarations in the internal DTD of Listing 4.1, it's time to see how they're referenced in the document instance. Remember that entities are referenced in different parts of the document instance, depending on their type:

- Parsed entities—either internal or external—such as those similar to the ADTEXT_1 and ADTEXT_2 entities are referenced exclusively in elements.

- Unparsed entities are referenced exclusively through an element's attribute.

In Listing 4.9 you can track down the first entity, `ADTEXT_1`, and see that it is referenced using the following notation: `&ADTEXT_1;`. The give-away is the ampersand (&) and semicolon (`;`) on either side of the entity name. When the XML processor encounters `&ADTEXT_1;`, it replaces the entity reference with the actual text used to define the entity (see Figure 4.5). The same holds true for the next entity, `&ADTEXT_2;`, which is also a parsed entity. The last entity, `ADGRAPHIC`, is referred to as a value for the `src` attribute in the `DISPLAYCONTENT` element. The rules of XML state that in order to use an unparsed entity in this way, not only must you declare the entity in the DTD, but you must declare that the attribute whose value will consist of the entity be declared as being an `ENTITY` type in the DTD. In the example in Listing 4.1, the following line of code accomplishes this in the internal DTD:

```
<!ATTLIST DISPLAYCONTENT src ENTITY #REQUIRED>
```

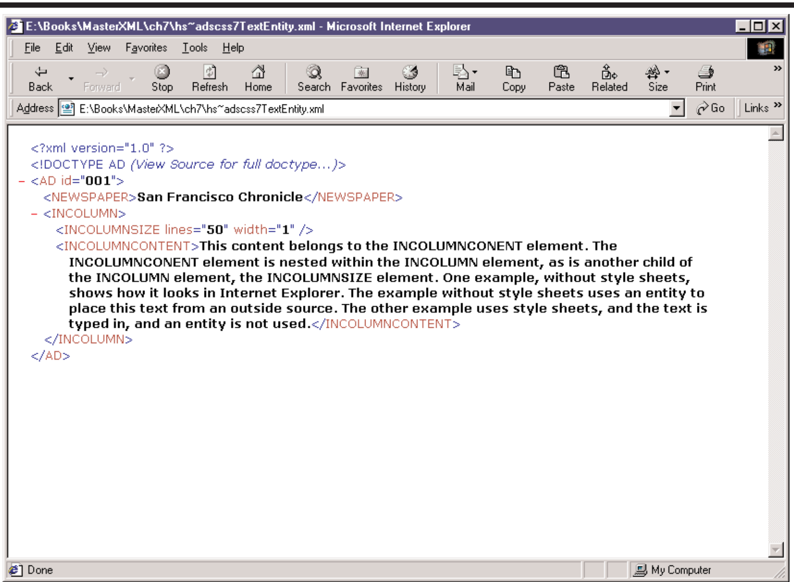


FIGURE 4.5 The &ADTEXT_1; entity is replaced by a text file when displayed in IE 5.

NOTE

The examples used here are for the humble beginnings of a hypothetical markup language for newspaper advertising. For a more robust, and real-world, example of a classified advertising DTD, see `www.zedak.com/admarkup`. The URL points to a system being developed by *The New York Times* and the Zedak Corporation for newspapers wanting to incorporate XML markup into older, more traditional classified advertising markup systems. The DTD is being considered as a standard by the Newspaper Association of America, and a portion of it appears in the next example.

Using Parameter Entities

You'll actually encounter *parameter entities* in more detail in Chapter 6, because they are used exclusively in DTDs. You should become familiar with parameter entities now, if only to begin the process of keeping the syntactical differences between parameter and general entities clear in your head.

Parameter entities accomplish the same thing as other entities that act as shortcuts. Using parameter entities, you can include element and attribute list declarations as groups and refer to them easily as single entities. You can even include an entire DTD in a parameter entity.

You can probably imagine, even if your knowledge of DTD development is still in an early stage, that quite a bit of time can be saved by using parameter entities in a DTD. Of course, it can be difficult to hard-code an XML document in a text editor if you have to validate it against a DTD that uses other DTDs, but in the long term that shouldn't be a major problem. This is because text editors that make developing against DTDs much easier are beginning to appear, and these tools will help you develop elements that follow the rules set forth in the DTD and, in turn, reduce the amount of time you spend manually searching through DTDs.

Parameter entities distinguish themselves from general entities by the inclusion of one simple character, the percent sign (%), in the entity declaration:

```
<!ENTITY % myParameterEntity
 "myElement | myElement2 | myElement3">
```

Notice that there is nothing else on either side of the percent sign. When a parameter entity is referenced, you simply place the percent sign next to the entity that is being referenced followed by a semicolon:

```
<!ELEMENT ANELEMENT
 (%myParameterEntity; |anotherElement)*>
```

A parameter entity is referenced only through another declaration within the DTD, never within the document instance. So you won't see anything like this:

```
<MYELEMENT>%anIllegalParameterEntity</MYELEMENT>
```

Actually, that's not entirely true. You may see something like the preceding line of code, but it won't mean anything as far as entities are concerned. An XML processor would simply parse the preceding code as it would any other character data.

A parameter entity can be defined with any valid DTD markup. This makes it very useful for large groups of entity declarations. You could catalog those entity declarations as part of a separate DTD, or within a text file, and include them using an external identifier like a system identifier or public identifier, like so:

```
<!ENTITY % myParameterEntity
 "http://www.myDomain.com/someEntities.txt">
```

The preceding line of code is an example of an external parameter entity. You could use such a parameter entity to manage large amounts of language encoding, for cataloging a library of images, or for e-commerce purposes. There are no limits.

TIP

Be kind to others. As your parameter entity references grow and the complexity of your DTDs increases, comment on the code you use to develop the parameter entities in your DTD as well.

Developing Parameter Entities

Parameter entity development is a bit beyond the scope of this chapter, because you haven't yet learned how to develop DTDs, but it's worthwhile to have a look at how they work in a real-world scenario. You can read Chapter 6 for a more thorough review of parameter entities. Listing 4.10 shows a portion of a DTD that manages a classified ad system for *The New York Times.*

Listing 4.10: Portions of a Classified Advertisement System for *The New York Times*

```
<!-- Copyright 1998, The New York Times.-->

<!--                           AD TEXT-->
<!ENTITY % inline
 "#PCDATA|font|glyph|image|keyword|mailbox|margin">
<!ENTITY % spacer "space|tab">
<!ENTITY % flow   "center|left|line|right">
<!ELEMENT text    (%inline;|%flow;|reply)*>
<!ELEMENT center  (%inline;|reply)*>
<!ELEMENT font    (%inline;|%flow;|reply)*>
<!ATTLIST font
            size (agate|5|6|10|12|13|14|18|24|30|31|
          36|48|60|72) "agate">
<!ELEMENT glyph   EMPTY>
<!ATTLIST glyph
            name (en|em|thin|figure|dash|open|close|
          1-8|3-8|5-8|7-8|1-4
          |3-4|1-3|2-3|1-2) #REQUIRED
>
<!ELEMENT keyword (#PCDATA)>
<!ATTLIST keyword
          format  CDATA    ""
          name    CDATA    #REQUIRED
          punct   CDATA    ""
          scale   CDATA    ""
>
<!ELEMENT left    (%inline;|reply)*>
<!ELEMENT line    (%inline;|%spacer;|reply)*>
```

Listing 4.10 shows a small portion of a much larger DTD. Without getting too hung up on the all the semantics of this DTD, see if you can find the way parameter entities are used. Let's go through that process step by step.

First, you can see an entity declaration early on in the listing for an `inline` entity. The code for this first entity declaration looks like this in Listing 4.10:

```
<!ENTITY % inline
"#PCDATA|font|glyph|image|keyword|mailbox|margin">
```

Notice the use of the percent sign (%), which tells the XML parser that the associated entity is a parameter entity. Also, notice that there's a space separating the percent sign from any other content. This is important, because it tells the XML processor that the entity is being declared and defined, not referenced. The declaration states that any element declared with the `inline` entity as part of its definition (either required or optional) would thus also consist of either PCDATA, or the font, glyph, image, keyword, mailbox, or margin elements.

To better understand this, take a look at an instance where the `inline` entity is actually used. You'll note that the `text` element declaration includes the `inline` entity in its definition:

```
<!ELEMENT text    (%inline;|%flow;|reply)*>
```

As you can see, the `inline` entity is used as part of the `text` element's definition, along with another entity and the `reply` element. This means that the `text` element could contain the `font` element, because the `font` element is declared in the `inline` definition. Unto itself `inline` is not an element; it's an entity and acts as a shortcut to gather a bunch of other declarations together for a more compact DTD.

Summary

As you've seen in this chapter, entities can be a powerful tool. They provide a convenient mechanism for managing large amounts of content. You can use them to protect your XML content from being interpreted as markup or for adding multilingual capability to your document. You can also use them as powerful shortcuts that can cut endless hours of coding. Databases may use them extensively to manage large catalogs of images, legal documents, technical specifications, and the like.

The next chapter will examine how to design your documents. XML, after you get comfortable with it, is not as complicated as it first seems. But designing documents for XML is an art unto itself. The next chapter will offer some insight into the XML document design process.