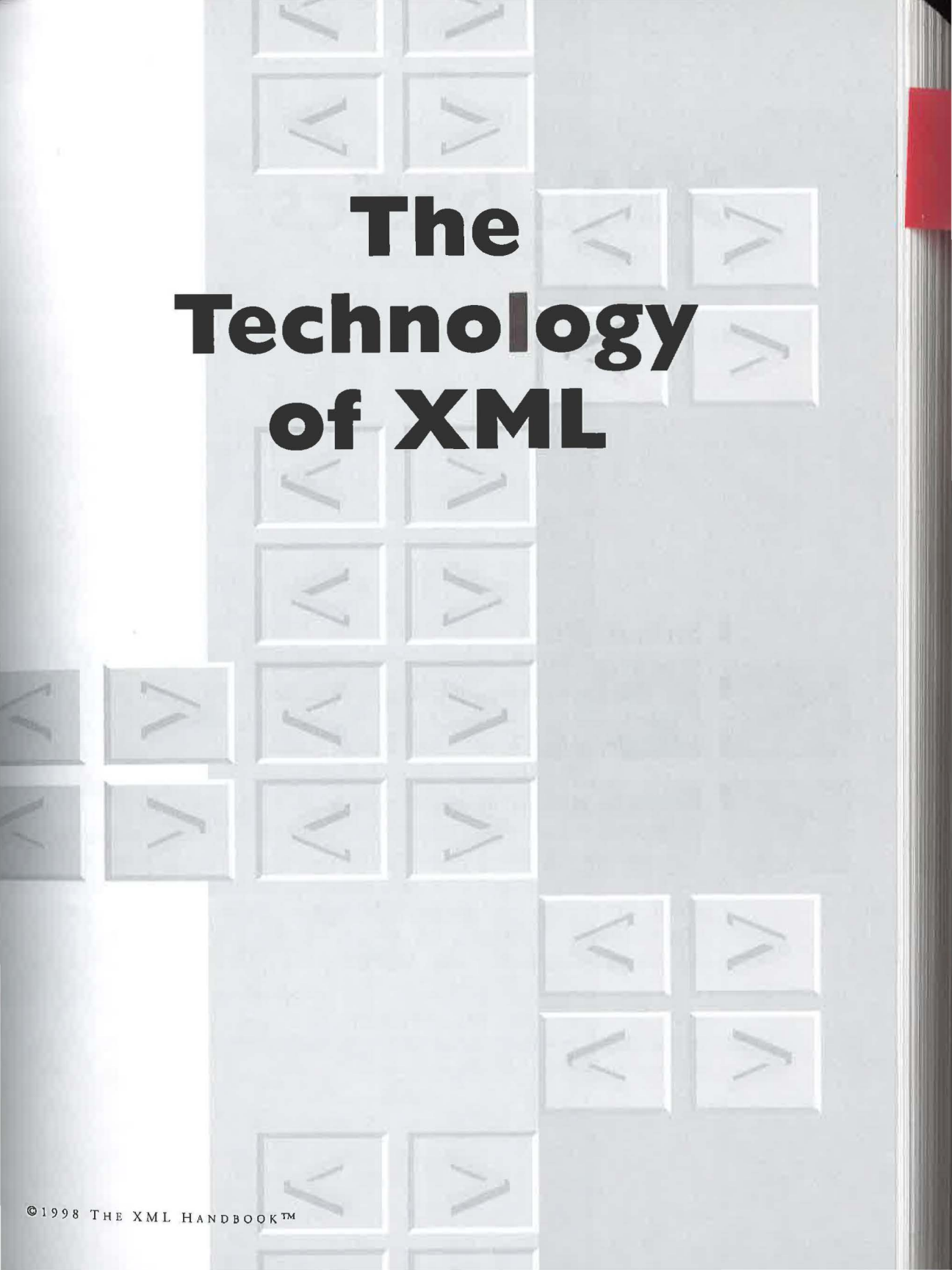
XML-Data

- DTD schemas
- Aliases
- Combining multiple schemas
- Datatypes

Part Five

- XML markup
- Document type definitions
- Linking and addressing
- Style sheets
- XML-Data
- Web Interface Definition Language (WIDL)



The Technology of XML

Creating a document type definition

- Document type declaration
- Element type declarations
- Attribute list declarations

Chapter 32

Creating your own document type definition is like creating your own markup language. If you have ever chafed at the limitations of a language with a fixed set of element types, such as HTML, TEI or LaTeX, then you will embrace the opportunity to create your own language.[1]

We should note again that it is possible to keep a document type definition completely in your head rather than writing the declarations for a DTD. Sometimes DTD designers do that while they are testing out ideas. Usually, though, you actually commit your ideas to declarations so that a validating processor can help you to keep your documents consistent.

Note also that, for the present, we are maintaining the distinction, discussed in 4.4.3, "Document type, DTD, and markup declarations", on page 61, between a document type, the XML markup rules for it (DTD), and the markup declarations that declare the DTD. Those *DTD declarations* are connected to the big kahuna of markup declarations – the *document type declaration.*

1. With its own set of limitations!

32.1 | Document type declaration

A document type declaration for a particular document might say "This document is a concert poster." The document type definition for the document would say "A concert poster must have the following features." As an analogy: in the world of art, you can *declare* yourself a practitioner of a particular movement, or you can *define* the movement by writing its manifesto.

The XML spec uses the abbreviation DTD to refer to document type definitions because we speak of them much more often than document type declarations. The DTD defines the allowed element types, attributes and entities and can express some constraints on their combination.

A document that conforms to its DTD is said to be *valid*. Just as an English sentence can be ungrammatical, a document can fail to conform to its DTD and thus be *invalid*. That does not necessarily mean, however, that it ceases to be an XML document. The word valid does not have its usual meaning here. An artist can fail to uphold the principles of an artistic movement without ceasing to be an artist, and an XML document can violate its DTD and yet remain a well-formed XML document.

As the document type declaration is optional, a well-formed XML document can choose not to declare conformance to any DTD at all. It cannot then be a valid document, because it cannot be checked for conformance to a DTD. It is not invalid, because it does not violate the constraints of a DTD.

XML has no good word for these merely well-formed documents. Some people call them "well-formed", but that is insufficiently precise. If the document were not well-formed, it would not be XML (by definition). Saying that a document is well-formed does not tell us anything about its conformance to a DTD at all.

For this reason, we prefer the terms used by the ISO for full-SGML: *type-valid*, meaning "valid with respect to a document type", and *non-type-valid*, the converse.

Example 32-1 is an XML document containing a document type declaration and document type definition for mailing labels, followed by an instance of the document type: a single label.

The document type declaration starts on the first line and ends with "]>". The DTD declarations are the lines starting with "<!ELEMENT". Those are *element type declarations*. You can also declare attributes, entities and notations for a DTD.

Example 32-1. XML document with document type declaration

```
<!DOCTYPE label[
    <!ELEMENT label (name, street, city, state, country, code)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT street (#PCDATA)>
    <!ELEMENT city (#PCDATA)>
    <!ELEMENT state (#PCDATA)>
    <!ELEMENT country (#PCDATA)>
    <!ELEMENT code (#PCDATA)>
]><label>
<name>Rock N. Robyn</name>
<street>Jay Bird Street</street>
<city>Baltimore</city>
<state>MD</state>
<country>USA</country>
<code>43214</code>
</label>
```

Recall from 3.4, "Entities: The physical structure", on page 38 that an XML document can be broken up into separate objects for storage, called "entities".[1] The document type declaration occurs in the first (or only) entity to be parsed, called the "document entity".

In Example 32-1, all of the DTD declarations that define the label DTD reside within the document entity. However, the DTD could have been partially or completely defined somewhere else. In that case, the document type declaration would contain a reference to another entity containing those declarations.

A document type declaration with only external DTD declarations looks like Example 32-2.

Example 32-2. Document type declaration with external DTD declarations

```
<?xml version="1.0"?>
<!DOCTYPE LABEL SYSTEM "http://www.sgmlsource.com/dtds/label.dtd">
<LABEL>
...
</LABEL>
```

They keyword SYSTEM is described more completely in 33.9.1, "System identifiers", on page 495. For now, we will just say that it tells the processor

1. Loosely, an entity is like a file.

to fetch some resource containing the external information. In this case, the external information is made up of the declarations that define the label DTD. They should be exactly the ones we had in the original label document. The big difference is that now they can be reused in hundreds, thousands, or even millions of label documents. Our simple DTD could be the basis for the largest junk mailing in history!

All document type declarations start with the string "<!DOCTYPE". Next they have the name of an element type that is defined in the DTD. The root element in the instance (described in 31.4, "Elements", on page 434) must be of the type declared in the document type declaration. If any of the DTD declarations are stored externally, the third part of the document type declaration must be either "SYSTEM" or "PUBLIC". We will cover "PUBLIC" later. If it is "SYSTEM", the final part must be a *URI* pointing to the external declarations. A URI is, for all practical purposes, a URL. URIs are discussed in 34.4, "Uniform Resource Identifier (URI)", on page 512.

Spec. Reference 32-1. DOCTYPE declaration

```
[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('['
                      (markupdecl | PEReference | S)* ']' S?)? '>'
[75] ExternalID ::=  'SYSTEM' S SystemLiteral
                    | 'PUBLIC' S PubidLiteral S SystemLiteral

[29] markupdecl ::= elementdecl | AttlistDecl | EntityDecl
                     | NotationDecl | PI | Comment
```

32.2 | Internal and external subset

In Example 32-1, the DTD declarations were completely *internal.* They were inside of the document type declaration. In Example 32-2, they were completely external. In many cases, there will be a mix of the two. This section will review these options and show how most XML document type declarations combine an internal part, called the *internal subset* and an external part, called the *external subset.*

From now on, as we'll almost always be writing about DTD declarations, we'll refer to them as "the DTD". We'll resort to the finer distinctions only when necessary for clarity.

We will start with another example of a DTD:

Example 32-3. Garage sale announcement DTD.

```
<!ELEMENT GARAGESALE (DATE, TIME, PLACE, NOTES)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT PLACE (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
```

These markup declarations would make up an ultra-simple DTD for garage sale announcements.[1] As you may have deduced, it declares five element types. We will get to the syntax of the declarations soon. First we will look at how they would be used. These could reside in a separate file called `garage.dtd` (for instance) and then every document that wanted to conform to them would declare its conformance using a document type declaration. This is shown in Example 32-4.

Example 32-4. Conforming garage sale document.

```
<!DOCTYPE GARAGESALE SYSTEM "garage.dtd">
<GARAGESALE>
<DATE>February 29, 1998</DATE>
<TIME>7:30 AM</TIME>
<PLACE>249 Cedarbrae</PLACE>
<NOTES>Lots of high-quality junk for sale!</NOTES>
</GARAGESALE>
```

Instead of a complete URL, we have just referred to the DTD's file name. Actually, this is still a URL. It is a relative URL. That means that in a standard Web server setup, the XML document entity and its DTD entity reside in the same directory. You could also refer to a full URL as we did in Example 32-2.

Example 32-5. Specifying a full URL

```
<!DOCTYPE GARAGESALE SYSTEM
          "http://www.tradestuff.com/stuff.dtd">
<GARAGESALE>
...
</GARAGESALE>
```

1. A garage sale is where North Americans spend their hard-earned money on other people's junk, which they will eventually sell at their own garage sales.

The relative URL is more convenient while you are testing because you do not need to have a full server installed. You can just put the two entities in the same directory on your hard drive. But your DTD and your instance can get even more cozy than sharing a directory. You can hoist your DTD into the same entity as the instance:

Example 32-6. Bringing a DTD into the instance

```
<!DOCTYPE GARAGESALE
[
<!ELEMENT GARAGESALE (DATE, PLACE, NOTES)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT PLACE (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
]>
<GARAGESALE>
...
</GARAGESALE>
```

The section between the square brackets is called the *internal subset* of the document type declaration. For testing, this is very convenient! You can edit the instance and the DTD without moving between entities. Since entities usually correspond to files, this means that instead of moving between two files, you need only edit one.

Although this is convenient, it is not great for reuse. The DTD is not available anywhere but in this file. Other documents cannot conform to this DTD without copying the declarations into their internal subset.

Often you will combine both approaches. Some of the DTD declarations can go in an external entity where it can be reused, and some of it can go in the same entity as the instance. Often graphic entities (see 33.6, "Unparsed entities", on page 486) would be declared in the internal subset because they are specific to a document. On the other hand, element type declarations would usually be in the *external subset*, the external part of the document type declaration:

Example 32-7. Reference to an external subset

```
<!DOCTYPE GARAGESALE SYSTEM "garage.dtd">
<!ENTITY LOGO SYSTEM "logo.gif">
]><GARAGESALE> ... </GARAGESALE>
```

The declarations in the internal subset are processed before those in the external subset. This gives document authors the opportunity to override[1] some kinds of declarations in the shared portion of the DTD.

Note that the content of both the internal subset and the external subset makes up the DTD. `garage.dtd` may have a `.dtd` extension but that is just a convention we chose to emphasize that the file contains DTD declarations. It is *not* necessarily the full set of them. The full set of DTD declarations is the combination of the declarations in the internal and external subsets.

Caution *Many people believe that the file containing the external subset is "the DTD". Until it is referenced from a document type declaration and combined with an internal subset (even an empty one) it is just a file that happens to have markup declarations in it. It is good practice, however, when an external subset is used, to restrict the internal subset to declarations that apply only to the individual document, such as entity declarations for graphics.*

It is often very convenient to point to a particular file and refer to it as "the DTD" for a given document type. As long as the concepts are straight in your mind, it does seem a trifle simpler than saying "the file that contains the markup declarations that I intend to reference as the external subset of the document type declaration for all documents of this type".

32.3 | Element type declarations

Elements are the foundation of XML markup. Every element in a valid XML document must conform to an element type declared in the DTD. Documents with elements that do not conform could be well-formed, but not valid. Here is an example of an element type declaration:

Element type declarations must start with the string "<!ELEMENT", followed by the name (or *generic identifier* of the element type being

1. Actually, preempt.

Example 32-8. Element type declaration.

```
<!ELEMENT memo (to, from, body )>
```

declared. Finally they must have a *content specification.* The content specification above states that elements of this type must contain a `to` element followed by a `from` element followed in turn by a `body` element. Here is the rule from the XML grammar:

Spec. Reference 32-2. Element type declaration

```
<!ELEMENT' S Name S contentspec S? '>'
```

Element type names are XML *names.* That means there are certain restrictions on the characters allowed in them. These are described in 31.1.4, "Names and name tokens", on page 428. Each element type declaration must use a different name because a particular element type cannot be declared more than once.

Caution *Unique element type declaration*
Unlike attribute and entities, element types can be declared only once.

32.4 | Element type content specification

Every element type has certain allowed content. For instance a document type definition might allow a `chapter` to have a `title` in its content, but would probably not allow a `footnote` to have a `chapter` in its content (though XML itself would not prohibit that!).

There are four kinds of content specification. These are described in Table 32-1.

Table 32-1 Content specification types

Content specification type	Allowed content
EMPTY content	May not have content. They are typically used for their attributes.
ANY content	May have any content at all.
Mixed content	May have character data or a mix of character data and sub-elements specified in mixed content specification.
Element content	May have only sub-elements specified in element content specification

32.4.1 *Empty content*

Sometimes we want an element type that can never have any content. We would give it a content specification of EMPTY. For instance an image element type like HTML's `img` would include a graphic from somewhere else. It would do this through an attribute and would not need any sub-elements or character data content. A cross-reference element type might not need content because the text for the reference might be generated from the target. A reference to an element type with the title "More about XML" might become "See *More about XML* on page 14".

You can declare an element type to have empty content by using the EMPTY keyword as the content specification:

Example 32-9. Empty element type

```
<!ELEMENT MY-EMPTY-ELEMENT EMPTY>
```

32.4.2 *ANY content*

Occasionally, you want an element type to be able to hold any element or character data. You can do this if you give it a content spec of ANY:

This is rarely done. Typically we introduce element type declarations to express the structure of our document types. An element type that has an

Example 32-10. Element type with ANY content.

```
<!ELEMENT LOOSEY-GOOSEY ANY>
```

ANY content specification is completely unstructured. It can contain any combination of character data and sub-elements. Still, ANY content element types are occasionally useful, especially while a DTD is being developed. If you are developing a DTD for existing documents, then you could declare each element type to have ANY content to get the document to validate. Then you could try to figure out more precise content specifications for each element type, one at a time.

32.4.3 *Mixed content*

Element types with *mixed content* are allowed to hold either character data alone or character data with child elements interspersed. A paragraph is a good example of a typical mixed content element. It might have character data with some mixed in emphasis and quotation sub-elements. The simplest mixed content specifications allow data only and start with a left parenthesis character ("("), followed by the string #PCDATA and a final close parenthesis (")"):

Example 32-11. Data-only mixed content.

```
<!ELEMENT emph (#PCDATA)>
<!ELEMENT foreign-language ( #PCDATA ) >
```

You may put white space between the parenthesis and the string #PCDATA if you like. The declarations above create element types that cannot contain sub-elements. Sub-elements that are detected will be reported as validity errors.

In other words, these elements do not really have "mixed" content in the usual sense. Like the word "valid", XML has a particular meaning for the word that is not very intuitive. Any content specification that contains #PCDATA is called mixed, whether sub-elements are allowed or not.

We can easily extend the DTD to allow a mix of elements and character data:

Example 32-12. Allow a mix of character data and elements

```
<!ELEMENT paragraph (#PCDATA|emph)*>
<!ELEMENT abstract (#PCDATA|emph|quot)*>
<!ELEMENT title ( #PCDATA | foreign-language | emph )* >
```

Note the trailing asterisks. They are required in content specifications that allow a mix of character data and elements. The reason that they are there will be clear when we study content models. Note also that we can put white space before and after the vertical bar ("|") characters.

These declarations create element types that allow a mix of character data and sub-elements. The element types listed after the vertical bars ("|"), are the allowed sub-elements. The following would be a valid `title` if we combine the declarations in Example 32-12 with those in Example 32-11

```
<title>this is a <foreign-language>tres gros</foreign-language>
      title for an <emph>XML</emph> book</title>
```

The `title` has character data ("This is a"), a `foreign-language` sub-element, some more character data ("title for an"), an `emph` sub-element and some final character data "book". We could have reordered the `emph` and `foreign-language` elements and the character data however we wanted. We could also have introduced as many (or as few) `emph` and `foreign-language` elements as we needed.

32.5 | Content models

The final kind of content specification is a "children" specification. This type of specification says that elements of the type can contain only child elements in its content. You declare an element type as having *element content* by specifying a *content model* instead of a mixed content specification or one of the keywords described above.

A content model is a pattern that you set up to declare what sub-element types are allowed and in what order they are allowed. A simple model for a `memo` might say that it must contain a `from` followed by a `to` followed by a `subject` followed by a `paragraph`. A more complex model for a `question-and-answer` might require `question` and `answer` elements to alternate.

A model for a `chapter` might require a single `title` element, one or two `author` elements and one or more `paragraphs`. When a document is vali-

dated, the processor would check that the element's content matches the model.

A simple content model could have a single sub-element type:

```
<!ELEMENT WARNING (PARAGRAPH)>
```

This says that a WARNING must have a single PARAGRAPH within it. As with mixed content specifications, you may place white space before or after the parentheses. We could also say that a WARNING must have a TITLE and then a PARAGRAPH within it:

```
<!ELEMENT WARNING (TITLE, PARAGRAPH)>
```

The comma (",") between the "TITLE" and "PARAGRAPH" GIs indicates that the "TITLE" must precede the "PARAGRAPH" in the "WARNING" element. This is called a *sequence*. Sequences can be as long as you like:

```
<!ELEMENT MEMO (FROM, TO, SUBJECT, BODY)>
```

You may put white space before or after the comma (",") between two parts of the sequence.

Sometimes you want to have a *choice* rather than a sequence. For instance a document type might be designed such that a FIGURE could contain either a GRAPHIC element (inserting an external graphic) or a CODE element (inserting some computer code).

```
<!ELEMENT FIGURE (GRAPHIC|CODE)>
```

The vertical bar character ("|") indicates that the author can choose between the elements. You can put white space before or after the vertical bar. You may have as many choices as you want:

```
<!ELEMENT FIGURE (CODE|TABLE | FLOW-CHART| SCREEN-SHOT)>
```

You may also combine choices and sequences using parenthesis. When you wrap parenthesis around a choice or sequence, it becomes a *content particle*. Individual GIs are also content particles. You can use any content particle where ever you would use a GI in a content model:

```
<!ELEMENT FIGURE (CAPTION, (CODE|TABLE|FLOW-CHART|SCREEN-SHOT) )>
<!ELEMENT CREATED ((AUTHOR | CO-AUTHORS), DATE )>
```

The content model for FIGURE is thus made up of a sequence of two content particles. The first content particle is a single element type name. The second is a choice of several element type names. You can break down the content model for CREATED in the same way.

You can make some fairly complex models this way. But when you write a DTD for a book, you do not know in advance how many chapters the book will have, nor how many paragraphs each chapter will contain. You need a way of saying that the part of the content specification that allows captions is *repeatable* – that you can match it many times.

Sometimes you will also want to make an element optional. For instance, some figures may not have captions. You may want to say that part of the specification for figures is optional.

XML allows you to specify that a content particle is optional or repeatable using an *occurrence indicator*. There are three occurrence indicators:

Table 32-2 Occurrence Indicators

Indicator	Content particle is...
?	Optional (0 or 1 time).
*	Optional and repeatable (0 or more times)
+	Required and repeatable (1 or more times)

Occurrence indicators directly follow a GI, sequence or choice. The occurrence indicator cannot be preceded by white space.

For instance we can make captions optional on figures:

```
<!ELEMENT FIGURE (CAPTION?, (CODE|TABLE|FLOW-CHART|SCREEN-SHOT))>
```

We can allow footnotes to have multiple paragraphs:

```
<!ELEMENT FOOTNOTE (P+)>
```

Because we used the "+" indicator, footnotes must have at least one paragraph. We could also have expressed this in another way:

```
<!ELEMENT FOOTNOTE (P, P*)>
```

This would require a leading paragraph and then 0 or more paragraphs following. That would achieve the same effect as requiring 1 or more paragraphs. The "+" operator is just a little more convenient than repeating the preceding content particle.

We can combine occurrence indicators with sequences or choices:

```
<!ELEMENT QUESTION-AND-ANSWER (INTRODUCTION,
                               (QUESTION, ANSWER)+,
                               COPYRIGHT?)>
```

It is also possible to make all of the element types in a content model optional:

```
<!ELEMENT IMAGE (CAPTION?)>
```

This allows the IMAGE element to be empty sometimes and not other times. The question mark indicates that CAPTION is optional. Most likely these IMAGE elements would link to an external graphic through an

attribute. The author would only provide content if he wanted to provide a caption.

In the document instance, empty IMAGE elements look identical to how they would look if IMAGE had been declared to be always empty. There is no way to tell from the document instance whether they were declared as empty or are merely empty in a particular case.

32.6 | Attributes

Attributes allow an author to attach extra information to the elements in a document. For instance a code element for computer code might have a lang attribute declaring the language that the code is in. On the other hand, you could also use a lang sub-element for the same purpose. It is the DTD designer's responsibility to choose a way and embody that in the DTD. Attributes have strengths and weaknesses that differentiate them from sub-elements so you can usually make the decision without too much difficulty.

The largest difference between elements and attributes is that attributes cannot contain elements and there is no such thing as a "sub-attribute". Attributes are always either text strings with no explicit structure (at least as far as XML is concerned) or simple lists of strings. That means that a chapter should not be an attribute of a book element, because there would be no place to put the titles and paragraphs of the chapter. You will typically use attributes for small, simple, unstructured "extra" information.

Another important difference between elements and attributes is that each of an element's attributes may be specified only once, and they may be specified in any order. This is often convenient because memorizing the order of things can be difficult. Elements, on the other hand, must occur in the order specified and may occur as many times as the DTD allows. Thus you must use elements for things that must be repeated, or must follow a certain pattern or order that you want the XML parser to enforce.

These technical concerns are often enough to make the decision for you. But if everything else is equal, there are some usability considerations that can help. One rule of thumb that some people use (with neither perfect success nor constant abject failure) is that elements usually represent data that is the natural content that should appear in every print-out or other rendition, Most formatting systems print out elements by default and do

not print out attributes unless you specifically ask for them. Attributes represent data that is of secondary importance and is often information about the information (*"metainformation"*).

Also, attribute names usually represent properties of objects, but element-type names usually represent parts of objects. So given a `person` element, sub-elements might represent parts of the body and attributes might represent properties like weight, height, and accumulated karma points.

We would advise you not to spend too much of your life trying to figure out exactly what qualifies as a part and what qualifies as a property. Experience shows that the question "what is a property?" ranks with "what is the good life?" and "what is art?". The technical concerns are usually a good indicator of the philosophical category in any event.

32.6.1 *Attribute-list declarations*

Attributes are declared for specific element types. You declare attributes for a particular element type using an *attribute-list declaration*. You will often see an attribute-list declaration right beside an element type declaration:

```
<!ELEMENT PERSON (#PCDATA)>
<!ATTLIST PERSON EMAIL CDATA #REQUIRED>
```

Attribute declarations start with the string "<!ATTLIST". Immediately after the white space comes an element type's generic identifier. After that comes the attribute's name, its *type* and its *default*. In the example above, the attribute is named `EMAIL` and is valid on `PERSON` elements. Its value must be *character data* and it is required – there is no default and the author must supply a value for the attribute on every `PERSON` element.

Spec. Reference 32-3. Attribute-list declarations

```
[52]  AttlistDecl ::=  '<!ATTLIST' S Name AttDef* S? '>'
[53]  AttDef ::=  S Name S AttType S DefaultDecl
```

You can declare many attributes in a single attribute-list declaration.[1]

You can also have multiple attribute-list declarations for a single element type:

1. That's why it is called a list!

Example 32-13. Declaring multiple attributes

```
<!ATTLIST PERSON EMAIL CDATA #REQUIRED
                 PHONE CDATA #REQUIRED
                 FAX CDATA #REQUIRED>
```

Example 32-14. Multiple declarations for one element type

```
<!ATTLIST PERSON HONORIFIC CDATA #REQUIRED>
<!ATTLIST PERSON POSITION CDATA #REQUIRED
                 ORGANIZATION CDATA #REQUIRED>
```

This is equivalent to putting the declarations altogether into a single attribute-list declaration.

It is even possible to have multiple declarations for the same attribute of the same element type. When this occurs, the first declaration of the attribute is binding and the rest are ignored. This is analogous to the situation with entity declarations.

Note that two different element types can have attributes with the same name without there being a conflict. Despite the fact that these attributes have the same name, they are in fact different attributes. For instance a SHIRT element could have an attribute SIZE that exhibits values SMALL, MEDIUM and LARGE and a PANTS element in the same DTD could have an attribute also named SIZE that is a measurement in inches:

```
     <!-- These are -->
<!ATTLIST SHIRT SIZE (SMALL|MEDIUM|LARGE) #REQUIRED>

     <!-- two different attributes -->
<!ATTLIST PANTS SIZE NUMBER #REQUIRED>
```

It is not good practice to allow attributes with the same name to have different semantics or allowed values in the same document. That can be quite confusing for authors.

32.6.2 *Attribute defaults*

Attributes can have *default values*. If the author does not specify an attribute value then the processor supplies the default value if it exists. A DTD designer can also choose not to supply a default.

Specifying a default is simple. You merely include the default after the type or list of allowed values in the attribute list declaration:

```
<!ATTLIST SHIRT SIZE (SMALL|MEDIUM|LARGE) MEDIUM>
<!ATTLIST SHOES SIZE NUMBER "13">
```

Any value that meets the constraints of the attribute list declaration is legal as a default value. You could not, however, use "abc" as a default value for an attribute with declared type `number` any more than you could do so in a start-tag in the document instance.

Sometimes you want to allow the user to omit a value for a particular attribute without forcing a particular default. For instance you could have an element `SHIRT` which has a `SIZE` attribute with a declared type of `NUMBER`. But some shirts are "one size fits all". They do not have a size. You want the author to be able to leave this value out and you want the processing system to *imply* that the shirt is "one size fits all". You can do this with an *impliable* attribute:

```
<!ATTLIST SHIRT SIZE NUMBER #IMPLIED>
```

The string "#IMPLIED" gives any processing program the right to insert whatever value it feels is appropriate. This may seem like a lot of freedom to give a programmer, but typically implied attributes are simply ignored. In the case of our `SHIRT`, there is no need to worry about "one size fits all" shirts because anybody can wear them. Authors should only depend upon the implied value when they do not care or where there is a well-defined convention of what the lack of a value "really" means. This is again a case of semantics and would be communicated to the author through some other document, DTD comment or other communication mechanism.

It is easy for an author to not specify a value for an attribute that is not required: just do not mention the attribute. Note that specifying an attribute value that is an empty string is *not* the same as not specifying an attribute value:

```
<SHIRT>         <!-- This conforms to the declaration above. -->
<SHIRT SIZE=""> <!-- This does *not* conform to the declaration. -->
```

The opposite situation to providing a default is where a document type designer wants to force the author to choose a value. If a value for an attribute is important and cannot reliably be defaulted, the designer can require authors to specify it with a *required* attribute default:

```
<!ATTLIST IMAGE URL CDATA #REQUIRED>
```

In this case, the DTD designer has made the `URL` attribute required on all `IMAGE` elements. This makes sense because without a URL to locate the image file, the `image` element is useless.

It may be surprising, but there are even times when it is useful to supply an attribute value that cannot be overriden at all. This is rare, but worth knowing about. Imagine, for instance, that an Internet directory maintainer

like *Yahoo*™ decides to write a robot [1] that will automatically extract the first section title of every document indexed by the directory. The difficulty is that different DTDs will have different element-type names for titles. HTML-like DTDs use `H1` etc. DocBook-like DTDs use `title`. TEI-like DTDs use `head`. Even if the robot knows about these DTDs, what about all of the others? There are potentially as many DTDs in existence as there are XML documents! It is not feasible to write a robot that can understand every document type.

The vendor needs to achieve some form of standardization. But it cannot force everyone to conform to the same DTD: that is exactly what XML is supposed to avoid! Instead, they can ask all document creators to label the elements that perform the *role of* section titles. They could do this with an attribute, such as `title-element`. The robot can then use the content of those elements to generate its index.

Each DTD designer thinks through the list of element types to add the attribute to. They specify what their element types mean in terms of the indexing system understood by the robot. They may not want authors changing the value on an element by element basis. They can prevent this with *fixed* attributes:

```
<!ATTLIST H1 TITLE-ELEMENT CDATA #FIXED "TITLE-ELEMENT">
<!ATTLIST HEAD TITLE-ELEMENT CDATA #FIXED "TITLE-ELEMENT">
<!ATTLIST TITLE TITLE-ELEMENT CDATA #FIXED "TITLE-ELEMENT">
```

Now all of the appropriate elements are marked with the attribute. No matter what else is in the DTD, the robot can find what it is looking for.

32.6.3 *Attribute types*

An important feature of attributes is that attributes have *types* that can enforce certain *lexical* and *semantic* constraints. *Lexical* constraints are constraints like "this attribute must contain only numerals". Semantic constraints are along the lines of "this attribute must contain the name of a declared entity". These constraints tend to be very useful in making robust DTDs and document processing systems.

However, it is vital to remember that **the value of an attribute is not necessarily the exact character string that you enter between the quotation marks**. That string first goes through a process called *attribute-*

1. A *robot* is an automatic Web information gatherer.

value normalization on its way to becoming the attribute value. Since attribute types apply to the *normalized value*, we had better digress for a moment to master normalization.

32.6.3.1 Attribute value normalization

XML processors normalize attribute values to make author's lives simpler. If it were not for normalization, you would have to be very careful where you put white space in an attribute value. For instance if you broke an attribute value across a line:

```
<GRAPHIC ALTERNATE-TEXT="This is a picture of a penguin
    doing the ritual mating dance">
```

You might do this merely becuse the text is too long for a single line in a text editor.

This sort of thing is normalized by the XML processor. The rules for this are a little intricate, but most times they will just do what you want them to. Let's look at them.

All XML attribute values are entered as quoted strings. They start and end with either single-quotes ("'") or double-quotes ("""). If you want to embed a single-quote character into an attribute value delimited by single quotes or a double-quote character into an attribute value delimited by double quotes, then you must use an entity reference as described in 31.7.1, "Predefined entities", on page 442.

The first thing the XML parser does to prepare for normalization is to strip off the surrounding quotes.

Then, character references are replaced by the characters that they reference. As we discussed earlier, character references allow you to easily insert "funny" characters.

Next, general entity references are replaced. This is important to note. While it is true that entity references are not allowed in markup, unnormalized attribute values are *text* – a mixture of markup and data. After normalization, only the data remains.[1]

If the expansion for an entity reference has another entity reference within it, that is expanded also, and so on and so forth. This would be rare in an entity used in an attribute value. After all, attribute values are usually

1. Philosophically, attribute values are metadata, but it is an article of faith in the XML world that metadata is data.

very short and simple. An entity reference in an attribute value cannot be to an external entity.

Newline characters in attribute values are replaced by spaces. *If* the attribute is known to be one of the tokenized types[1] (see below), then the parser must further remove leading and trailing spaces. So " token " becomes "token". It also collapses multiple spaces between tokens into a single space, so that "space between" would become "space between". The distinction between *unnormalized attribute value text* and *normalized attribute value data* trips up even the experts. Remember, when reading about attribute types, that they apply to the normalized data, not the unnormalized text.

32.6.3.2 CDATA and name token attributes

The simplest type of attribute is a *CDATA* attribute. The CDATA stands for "character data". The declaration for such an attribute looks like this:

Example 32-15. CDATA Attributes

```
<!DOCTYPE ARTICLE[
<!ELEMENT ARTICLE>
<!ATTLIST ARTICLE DATE CDATA #REQUIRED>
...
]>
<ARTICLE DATE="January 15, 1999">
...
</ARTICLE>
```

Character data attribute values can be any string of characters. Basically anything else is legal in this type of attribute value.

Name token (NMTOKEN) attributes are somewhat like CDATA attributes. The biggest difference is that they are restricted in the characters that name tokens allow. Name tokens were described in 31.1.4, "Names and name tokens", on page 428. To refresh your memory, they are strings made up of

1. If, in other words, attribute-list declarations were provided and the processor is either a validating processor or a non-validating processor that decides to read them.

letters, numbers and a select group of special characters: period (“.”), dash (“-”), underscore (“_”) and colon (“:”).

Example 32-16. Name token attribute type

```
<!DOCTYPE PARTS-LIST[
...
<!ATTLIST PART DATE NMTOKEN #REQUIRED>
...
]>
<PARTS-LIST>
...
<PART DATE="1998-05-04">...</PART>

</PARTS-LIST>
]>
```

An empty string is not a valid name token, whereas it would be a valid CDATA attribute value.

Name tokens can be used to allow an attribute to contain numbers that need special characters. They allow the dash, which can be used as a minus sign, the period, which can be a decimal point, and numbers. These are useful for fractional and negative numbers. You can also use alphabetic characters to specify units.

Name tokens can also be used for naming things. This is similar to how you might use variable names in a programming language. For instance, if you used XML to describe the structure of a database, you might use name tokens to name and refer to fields and tables. The restrictions on the name token attribute type would prevent most of the characters that would be illegal in field and table names (spaces, most forms of puncutation, etc.). If there is a reason that all fields or record names must be unique, then you would instead use the *ID* attribute type discussed in 32.6.3.4, “ID and IDREF attributes”, on page 470.

If it is appropriate to have more than one name token, then you can use the NMTOKENS attribute type which stands for “name tokens”. For instance in describing a database:

One other difference between CDATA attributes and NMTOKEN attributes is in their *normalization*. This was discussed in 32.6.3.1, “Attribute value normalization”, on page 467.

Example 32-17. Name tokens attribute type

```
<!DOCTYPE DATABASE [
...
<!ELEMENT TABLE EMPTY>
<!ATTLIST TABLE NAME NMTOKEN #REQUIRED
                FIELDS NMTOKENS #REQUIRED>
...
]>
<DATABASE>
...
<TABLE NAME="SECURITY" FIELDS="USERID PASSWORD DEPARTMENT">
...
</DATABASE>
```

32.6.3.3 Enumerated and notation attributes

Sometimes as a DTD designer you want to create an attribute that can only exhibit one of a short list of values: "small/medium/large", "fast/slow"; "north/south/east/west". *Enumerated attribute types* allow this. In a sense, they provide a choice or menu of options.

The syntax is reminiscent of choice lists in element type declarations:

```
<!ATTLIST CHOICE (OPTION1|OPTION2|OPTION3) #REQUIRED>
```

You may provide as many choices as you like. Each choice is an XML *name token* and must meet the syntactic requirements of name tokens described in 31.1.4, "Names and name tokens", on page 428.

There is another related attribute type called a notation attribute. This attribute allows the author to declare that the element's content conforms to a declared notation. Here is an example involving several ways of representing dates:

```
<!ATTLIST DATE NOTATION (EUROPEAN-DATE|US-DATE|ISO-DATE) #REQUIRED>
```

In a valid document, each notation allowed must also be declared with a notation declaration.

32.6.3.4 ID and IDREF attributes

Sometimes it is important to be able to give a name to a particular occurrence of an element type. For instance, to make a simple hypertext link or cross-reference from one element to another, you can name a particular section or figure. Later, you can refer to it by its name. The target element is

labeled with an *ID* attribute. The other element refers to it with an *IDREF* attribute. This is shown in Example 32-18

Example 32-18. ID and IDREF used for cross-referencing

```
<!DOCTYPE BOOK [
...
<!ELEMENT SECTION (TITLE, P*)>
<!ATTLIST SECTION MY-ID ID #IMPLIED>
<!ELEMENT CROSS-REFERENCE EMPTY>
<!ATTLIST CROSS-REFERENCE TARGET IDREF #REQUIRED>
...
]>
<BOOK>
...
<SECTION MY-ID="Why.XML.Rocks"><TITLE>Features of XML</TITLE>
...
</SECTION>

...
If you want to recall why XML is so great, please see
the section titled <CROSS-REFERENCE TARGET="Why.XML.Rocks"/>.
...
</BOOK>
```

The style sheet would instruct browsers and formatters to replace the cross-reference element with the name of the section. This would probably be italicized and hyperlinked or labeled with a page number if appropriate.

Note that we made the section's MY-ID optional. Some sections will not need to be the target of a cross-reference, hypertext link or other reference and will not need to be uniquely identified. The TARGET attribute on CROSS-REFERENCE is required. It does not make sense to have a cross-reference that does not actually refer to another element.

IDs are XML names, with all of the constraints described in 31.1.4, "Names and name tokens", on page 428. Every element can have at most one ID, and thus only one attribute per element type be an ID attribute. All IDs specified in an XML document must be unique. A document with two ID attributes whose values are the same is invalid. Thus "chapter" would not be a good name for an ID, because it would make sense to use it in many places. "introduction.chapter" would be a logical ID because it would uniquely identify a particular chapter.

IDREF attributes must refer to an element in the document. You may have as many IDREFs referring to a single element as you need. It is also

possible to declare an attribute that can potentially exhibit more than one IDREF by declaring it to be of type IDREFS:

```
<!ATTLIST RELATED-CHAPTERS TARGETS IDREFS #REQUIRED>
```

Now the TARGETS attribute may have one or more IDREFs as its value. There is no way to use XML to require that an attribute take two or more, or three or more, (etc.) IDREFs. You will recall that we could do that sort of thing using content models in element type declarations. There is no such thing as a content model for attributes. You could model this same situation by declaring RELATED-CHAPTERS to have content of one or more or two or more (etc.) CHAPTER-REF elements that each have a single IDREF attribute (named TARGET in this example):

Example 32-19. IDREF attributes

```
<!DOCTYPE BOOK[
...
<!ELEMENT RELATED-CHAPTERS (CHAPTER-REF+)>
<!ELEMENT CHAPTER-REF EMPTY>
<!ATTLIST CHAPTER-REF TARGET IDREF #REQUIRED>
...
]>
<BOOK>
...
<RELATED-CHAPTERS>
<CHAPTER-REF TARGET="introduction.to.xml">
<CHAPTER-REF TARGET="xml.rocks">
</RELATED-CHAPTERS>
...
</BOOK>
```

As you can see, element type declarations have the benefit of having content models, which can define complex structures, and attributes have the benefit of attribute types, which can enforce lexical and semantic constraints. You can combine these strengths to make intricate structures when this is appropriate.

32.6.3.5 ENTITY attributes

External unparsed entities are XML's way of referring to objects (files, CGI script output, etc.) on the Web that should not be parsed according to XML's rules. Anything from HTML documents to pictures to word proces-

sor files fall into this category. It is possible to refer to unparsed entities using an attribute with declared type ENTITY. This is typically done either to hyperlink to, reference or include an external object:

Example 32-20. Entity attribute type

```
<!DOCTYPE ARTICLE[
<!ATTLIST BOOK-REF TARGET ENTITY #REQUIRED>
...
<!ENTITY another-book SYSTEM
              "http://www.buyOurBooks.com/TheOtherBook.html">
...
]><BOOK>
...
<BOOK-REF target="another-book">
...
</BOOK>
```

You can also declare an attribute to be of type *ENTITIES*, in which case its value may be the name of more than one entity. It is up to the application or stylesheet to determine whether a reference to the entity should be treated as a hot link, embed link or some other kind of link. The processor merely informs the application of the existence and notation of the entity. You can find information on unparsed entities and notations in Chapter 33, "Entities: Breaking up is easy to do", on page 476 and 32.7, "Notation Declarations", on page 474.

32.6.3.6 Summary of attribute types

There are two *enumerated* attribute types: *enumeration* attributes and NOTATION attributes.

Seven attribute types are known as *tokenized* types because each value represents either a single token (ID, IDREF, ENTITY, NMTOKEN) or a list of tokens (IDREFS, ENTITIES, and NMTOKENS).

The final type is the CDATA string type which is the least constrained and can hold any combination of XML characters as long as "special characters" (the quote characters and ampersand) are properly entered.

Table 32-3 Summary of attribute types

Type	Lexical constraint	Semantic constraint
CDATA	None	None
Enumeration	Nmtoken	Must be in the declared list.
NOTATION	Name	Must be in the declared list and a declared notation name.
ID	Name	Must be unique in document.
IDREF	Name	Must be some element's ID.
IDREFS	Names	Must each be some element's ID.
ENTITY	Name	Must be a declared entity name.
ENTITIES	Names	Must each be a declared entity name.
NMTOKEN	Name Token	None
NMTOKENs	Name Tokens	None

32.7 | Notation Declarations

Notations are referred to in various parts of an XML document, for describing the data content notation of different things. A data content notation is the definition of how the bits and bytes of class of object should be interpreted. According to this definition, XML is a data content notation, because it defines how the bits and bytes of XML documents should be interpreted. Your favorite word processor also has a data content notation. The notation declaration gives an internal name to an existing notation so that it can be referred to in attribute list declarations, unparsed entity declarations, and processing instructions.

The most obvious place that an XML document would want to describe the notation of a data object is in a reference to some other resource on the web. It could be an embedded graphic, an MPEG movie that is the target of a hyperlink, or anything else. The XML facility for linking to these data resources is the entity declaration, and as we discussed earlier, they are referred to as *unparsed entities*. Part of the declaration of an unparsed entity is the name of a declared notation that provides some form of pointer to the

external definition of the notation. The external definition could be a public or system identifier for documentation on the notation, some formal specification or a helper application that can handle objects represented in the notation.

Example 32-21. Notations for unparsed entities

```
<!NOTATION HTML SYSTEM "http://www.w3.org/Markup">
<!NOTATION GIF SYSTEM "gifmagic.exe">
```

Another place that notations arise are in the notation attribute type. You use this attribute type when you want to express the notation for the data content of an XML element. For instance, if you had a date element that used ISO or EU date formats, you could declare notations for each format:

Example 32-22. Notations for unparsed entities

```
<!NOTATION ISODATE PUBLIC +//APPROPRIATE-IDENTIFIER-HERE//>
<!NOTATION EUDATE PUBLIC +//APPROPRIATE-IDENTIFIER-HERE//>
<!ELEMENT TODAY (#PCDATA)>
<!ATTLIST TODAY DATE-FORMAT NOTATION (ISODATE|EUDATE) #REQUIRED>
```

Now the DATE-FORMAT attribute would be restricted to those two values, and would thus signal to the application that the content of the TODAY element conforms to one or the other.

Finally, notations can be used to give XML names to the targets for processing instructions. This is not strictly required by XML, but it is a good practice because it provides a sort of documentation for the PI and could even be used by an application to invoke the target.

This seems like a good way to close this chapter. DTDs are about improving the permanence, longevity, and wide reuse of your data, and the predictability and reliability of its processing. If you use them wisely, they will save you time and money.

Tip *Learning the syntax of markup declarations so that you can write DTDs is important, but learning how to choose the right element types and attributes for a job is a subtle process that requires a book of its own. We suggest David Megginson's Structuring XML Documents, also in this series (ISBN 0-13-642299-3).*

Entities: Breaking up is easy to do

- Parameter and general
- Internal and external
- Parsed and unparsed

Chapter

33

XML allows flexible organization of document text. The XML constructs that provide this flexibility are called *entities*. They allow a document to be broken up into multiple storage objects and are important tools for reusing and maintaining text.

33.1 | Overview

In simple cases, an entity is like an abbreviation in that it is used as a short form for some text. We call the "abbreviation" the *entity name* and the long form the *entity content*. That content could be as short as a character or as long as a chapter. For instance, in an XML document, the entity `dtd` could have the phrase "document type definition" as its content. Using a reference to that entity is like using the word DTD as an abbreviation for that phrase – the parser replaces the reference with the content.

You create the entity with an *entity declaration*. Here is an entity declaration for an abbreviation:

Entities can be much more than just abbreviations. There are several different kinds of entities with different uses. We will first introduce the differ-

Example 33-1. Entity used as an abbreviation

```
<!ENTITY dtd "document type definition">
```

ent variants in this overview and then come back and describe them more precisely in the rest of the chapter. We approach the topic in this way because we cannot discuss the various types of entity entirely linearly. Our first pass will acquaint you with the major types and the second one will tie them together and provide the information you need to actually use them.

Another way to think of an entity is as a box with a label. The label is the entity's name. The content of the box is some sort of text or data. The entity declaration creates the box and sticks on a label with the name. Sometimes the box holds XML text that is going to be parsed (interpreted according to the rules of the XML notation), and sometimes it holds data, which should not be.

If the content of an entity is XML text that the processor should parse, the XML spec calls it a *parsed entity.* The name is badly chosen because it is, in fact, unparsed; it will be parsed only if and when it is actually used.

If the content of an entity is data that is not to be parsed, the XML spec calls it an unparsed entity. This name isn't so great either because, as we just pointed out, an XML text entity is also unparsed.

We'll try to minimize the confusion and to avoid saying things like "a parsed entity will be parsed by the XML parser". But we sure wish they had named them "text entity" and "data entity".

The abbreviation in Example 33-1 is a parsed entity. Parsed entities, being XML text, can also contain markup. Here is a declaration for a parsed entity with some markup in it:

Example 33-2. Parsed entity with markup

```
<!ENTITY dtd "<term>document type definition</term>">
```

The processor can also fetch content from somewhere on the Web and put that into the box. This is an *external* entity. For instance, it could fetch a chapter of a book and put it into an entity. This would allow you to reuse the chapter between books. Another benefit is that you could edit the chapter separately with a sufficiently intelligent editor. This would be very useful if you were working on a team project and wanted different people to work on different parts of a document at once.

Example 33-3. External entity declaration

```
<!ENTITY intro-chapter SYSTEM "http://www.megacorp.com/intro.xml">
```

Entities also allow you to edit very large documents without running out of memory. Depending on your software and needs, either each volume or even each article in an encyclopedia could be an entity.

An author or DTD designer refers to an entity through an *entity reference*. The XML processor replaces the reference by the content, as if it were an abbreviation and the content was the expanded phrase. This process is called *inclusion*. After the operation we say either that the entity reference has been *replaced* by the entity content or that the entity content has been *included*. Which you would use depends on whether you are talking from the point of view of the entity reference or the entity content. The content of parsed entities is called their *replacement text*.

Here is an example of a parsed entity declaration and its associated reference:

Example 33-4. Entity Declaration

```
<!DOCTYPE MAGAZINE[
...
<!ENTITY title "Hacker Life">
...
]>
<MAGAZINE>
<TITLE>&title;</TITLE>
...
<P>Welcome to the introductory issue of &title;. &title; is
geared to today's modern hacker.
...
</MAGAZINE>
```

Anywhere in the document instance that the entity reference "&title;" appears, it is *replaced* by the text "Hacker Life". It is just as valid to say that "Hacker Life" is *included* at each point where the reference occurs. The ampersand character starts all general entity references and the semicolon ends them. The text between is an entity name.

Spec. Reference 33-1. General entity reference

```
[68]  EntityRef ::=  '&' Name ';'
```

We have looked at entities that can be used in the creation of XML documents. Others can only be used to create XML DTDs. The ones we have been using all along are called *general* entities. They are called general entities because they can generally be used anywhere in a document. The ones that we use to create DTDs are called *parameter* entities.

We would use parameter entities for most of the same reasons that we use general entities. We want document type definitions to share declarations for element types, attributes and notations, just as we want documents to share chapters and abbreviations. For instance many DTDs in an organization might share the same definition for a paragraph element type named *para*. The declaration for that element type could be bundled up with other common DTD components and used in document type definitions for memos, letters and reports. Each DTD would include the element type declaration by means of a parameter entity reference.

Unparsed entities are for holding data such as images or molecular models in some data object notation. The application does not expect the processor to parse that information because it is not XML text.

Although it is an oversimplification, it may be helpful in your mind to remember that unparsed entities are often used for pictures and parsed entities are usually used for character text. You would include a picture through an unparsed entity, since picture representations do not (usually!) conform to the XML specification. Of course there are many kinds of non-XML data other than graphics, but if you can at least remember that unparsed entities are used for graphics then you will remember the rest also.

Example 33-5. Unparsed entity declaration

```
<!ENTITY picture SYSTEM "http://www.home.org/mycat.gif" NDATA GIF>
```

We use unparsed entities through an entity attribute. A processor does not expand an entity attribute, but it tells the application that the use occurred. The application can then do something with it. For instance, if the application is a Web browser, and the entity contains a graphic, it could display the graphic. Entity attributes are covered in 32.6.3.5, "ENTITY attributes", on page 472.

33.2 | Entity details

Caution *Like other names in XML, entity names are case-sensitive: &charles; refers to a different entity from &Charles;.*

It is good that XML entity names are case-sensitive because they are often used to name letters. Case is a convenient way of distinguishing the upper-case version of a letter from the lower-case one. "Sigma" would represent the upper-case version of the Greek letter, and "sigma" would be the lower-case version of it. It would be possible to use some other convention to differentiate the upper- and lower-case versions, such as prefixes. That would give us "uc-Sigma" and "lc-Sigma".

Entities may be declared more than once, but only the first declaration is *binding*. All subsequent ones are ignored as if they did not exist.

```
<!ENTITY abc "abcdefghijklmnopqrst"> <!-- This is binding. -->
<!ENTITY abc "ABCDEFGHIJKLMNOPQRST"> <!-- This is ignored. -->
<!ENTITY abc "AbCdEfGhIjKlMnOpQrSt"> <!-- So is this.       -->
```

Declarations in the internal DTD subset are processed before those in the external subset, as described in Chapter 32, "Creating a document type definition ", on page 448. In practice, document authors can override parameter entities in the external subset of the DTD by declaring entities of the same name in the internal subset.

Entities are not difficult to use, but there are several variations and details that you should be aware of. We have already covered the major varieties, but only informally.

There is one special entity, called the *document entity* which is not declared, does not have a name and cannot be referenced. The document entity is the entity in which the processor started the current parse. Imagine you download a Web document called `catalog.xml`. Before a browser can display it, it must start to parse it, which makes it the document entity. It may include other entities, but because parsing started with `catalog.xml`, those others are not the document entity. They are just ordinary external entities.

If you click on a link and go to another XML Web page, then the processor must parse that page before it can display it. That page is the document entity for the new parse. In other words, even the simplest XML document

has at least one entity: the document entity. The processor starts parsing the document in the document entity and it also must finish there.[1]

The document entity is also the entity in which the XML declaration and document type declaration can occur.

You may think it is strange for us to call this an entity when it is not declared as such, but if we were talking about files, it would probably not surprise you. It is common in many computer languages to have files that include other files. Even word processors allow this. We will often use the word entity to refer to a concept analogous to what you would think of as a file, although entities are more flexible. Entities are just "bundles of information". They could reside in databases, zip files, or be created on the fly by a computer program.

33.3 | Classifications of entities

There are many interesting things that you can do with entities. Here are some examples:

- You could store every chapter of a book in a separate file and link them together as entities.
- You could "factor out" often-reused text, such as a product name, into an entity so that it is consistently spelled and displayed throughout the document.
- You could update the product name entity to reflect a new version. The change would be instantly visible anywhere the entity was used.
- You could create an entity that would represent "legal boilerplate" text (such as a software license) and reuse that entity in many different documents.
- You could integrate pictures and multimedia objects into your document.
- You could develop "document type definition components" that could be used in many document type definitions. These would allow you to reuse the declarations for common

1. To put it mystically: it is the alpha and the omega of entities.

element types (such as paragraph and emphasis) across several document types.

Because XML entities can do so many things, there are several different varieties of them. But XML entities do not break down into six or eight different types with simple names. Rather, you could think of each entity as having three properties that define its type. This is analogous to the way that a person could be tall or short and at the same time male or female and blonde or brunette.

Similarly, entities can be *internal* or *external*, *parsed* or *unparsed* and *general* or *parameter*. There is no single word for a short, male, brunette, and there is similarly no single word for an internal, parsed, parameter entity.

Caution *Some combinations of entity types are impossible. Obviously an entity cannot be both internal and external, just as a person could not be both blonde and brunette. It turns out that due to restrictions on unparsed entities, there are five combinations that are valid and three that are not.*

Most of the rest of this chapter will describe the five types of entities in greater depth. We will use one convention that might be confusing without this note. In a section on, for instance, internal parsed general entities, we may describe a constraint or feature of all general entities. When we do so, we will use the word "general entity" instead of "internal general entity". This convention will allow us to avoid repeating text that is common among entity types. We will refer back to that text from other sections when it becomes relevant.

33.4 | Internal general entities

Internal parsed general entities are the simplest type of entity. They are essentially abbreviations defined completely in the document type declaration section of the XML document.

All internal general entities are parsed entities. This means that the XML processor parses them like any other XML text. Hence we will leave out the redundant word "parsed" and refer to them simply as internal general entities.

The content for an internal general entity is specified by a string literal after the entity's name. The string literal may contain any markup, including references to other entities. An example is in Example 33-6.

Example 33-6. Internal general entity

```
<?xml version="1.0"?>
<!DOCTYPE EXAMPLE SYSTEM "example.dtd"[
    <!ENTITY xml "Extensible Markup Language">
]>
<EXAMPLE>
    &xml;
</EXAMPLE>
```

Internal general entities can be referenced anywhere in a document instance. They can also be referenced in the content of another general entity. Because they are general entities, they cannot be used to hold markup declarations for expansion in the DTD. They can only hold document content. Because of this, Example 33-7 is not well-formed.

Example 33-7. Illegal: General entities cannot be reference in the DTD

```
<?xml version="1.0"?>
<!DOCTYPE EXAMPLE[
    <!ENTITY xml "Extensible Markup Language">
    &xml;
]>
```

The grammar rules for internal general entities are described in Specification reference 33-2.

Spec. Reference 33-2. Internal general entities

```
[70]  EntityDecl ::=  GEDecl | PEDecl
[71]  GEDecl ::=  '<!ENTITY' S Name S EntityDef S? '>'
[73]  EntityDef ::=  EntityValue | (ExternalID NDataDecl?)
[9]  EntityValue ::=  '"' ([^%&"] | PEReference | Reference)* '"'
                    | "'" ([^%&'] | PEReference | Reference)* "'"
```

33.5 | External parsed general entities

Every XML entity is either internal or external. The content of internal entities occurs right in the entity declarations. External entities get their content from somewhere else in the system. It might be another file on the hard disk, a Web page or an object in a database. Wherever it is, it is located through an *external identifier*. Usually this is just the word SYSTEM followed by a URI (see 34.4, "Uniform Resource Identifier (URI)", on page 512).

In this section, we are interested specifically in external parsed general entities. Here is an example of such an entity:

```
<!ENTITY ent SYSTEM "http://www.house.gov/Constitution.xml">
```

It is the keyword SYSTEM that tells the processor that the next thing in the declaration is a URI. The processor gets the entity's content from that URI. The combination of SYSTEM and the URI is called an external identifier because it identifies an external resource to the processor. There is another kind of external identifier called a PUBLIC identifier. It is denoted by the keyword PUBLIC. External identifiers are described in 33.9, "External identifiers", on page 494

External parsed general entities can be referenced in the same places that internal general entities can be – the document instance and the replacement text of other general entities – except not in the value of an attribute.

33.5.1 *External parsed entity support is optional*

XML processors are allowed, but not required, to validate an XML document when they parse it. The XML specification allows a processor that is not validating a document to completely ignore declarations of external parsed entities (both parameter and general). There is no way to control this behavior with the standalone document declaration or any other XML markup.

The reason for this is improved Web surfing performance. The XML working group thought that it was important for processors to be able to download the minimum amount of data required to do their job and no more. For instance, a browser could display unresolved external parsed entities as hypertext links that the user could click on to receive. Because the

entity would only be downloaded on demand, the original page might display faster.

Unfortunately this is very inconvenient for authors, because it means that external parsed entities are essentially unreliable in systems that you do not completely control (e.g. the Internet vs. an intranet).

Caution *External parsed entity processing is optional XML processors can ignore external parsed entities. If you use them to store parts of your documents, those parts will only show up at the browser vendor's option.*

In practice this probably means that you should not put documents that use external entities on the Web until a pattern for browser behavior emerges. In the meantime, tools like James Clark's sgmlnorm (part of SP) (see 400) can read an XML document that uses external entities and expand all of the entities for you. Hopefully future versions of the XML specification will make external entity inclusion mandatory.

33.6 | Unparsed entities

Every XML entity is either an *unparsed*<> entity or a *parsed*<> entity. Unparsed entities external entities that the XML processor does not have to parse. For example a graphic, sound, movie or other multimedia object would be included through an unparsed entity. You can imagine the number of error messages you would get if an XML processor tried to interpret a graphic as if it were made up of XML text!

It is occasionally useful to refer to an XML document through an unparsed entity, as if it were in some unparsable representation. You might embed a complete letter document in a magazine document in this way. Rather than extending the magazine DTD to include letter elements, you would refer to it as an unparsed entity. Conceptually, it would be handled in the same way a picture of the letter would be handled. If you refer to it as an unparsed entity, the processor that handles the magazine does not care that the letter is actually XML.

All unparsed entities are external entities because there is no way to xpress non-XML information in XML entities. They are also all general ntities because it is forbiddent (and senseless) to embed data in XML)TDs. Hence, the term "unparsed entity" implies the terms "general" and external".

Syntactically, declarations of unparsed entities are differentiated from hose of other external entities by the keyword NDATA followed by a *notation* ıame.

pec. Reference 33-3. Non-XML data declaration

```
DataDecl ::=  S 'NDATA' S Name
```

The name at the end is the name of a declared notation. Notation decla- ations are described in 32.7, "Notation Declarations", on page 474. The processor passes this to the application as a hint about how the application hould approach the entity.

If the application knows how to deal with that sort of entity (for instance f it is a common graphics notation) then it could do so directly. A browser night embed a rendition of the entity. It might also make a hyperlink to the ntity. If it needs to download or install some other handler such as a Java program or Active-X control, then it could do so. If it needs to ask the user vhat to do it could do that also. The XML specification does not say what t must do. XML only expects processors to tell applications what the declared notation is and the applications must figure out the rest.

In the rare case that the entity is an XML document, the application night decide to process it, create a rendition of it, and then embed it. Alter- natively, it might decide to make a hyperlink to it.

33.7 | Internal and external parameter entities

XML entities are classified according to whether they can be used in the DTD or in the document instance. Entities that can only be used in the DTD are called *parameter* entities. For instance, you might want to wrap

up a few declarations for mathematical formuale element types and reuse the declarations from DTD to DTD.

The other entities can be used more generally (throughout the entire document instance), and are called *general* entities. Authors can use general entities as abbreviations, for sharing data among documents, including pictures, and many similar tasks.

There is an important reason why the two types are differentiated. When authors create documents, they want to be able to choose entity names without worrying about accidently choosing a name that was already used by the DTD designer. If there were no distinction between entities specific to the DTD and general to the document instance, according to XML's rules, the first declaration would win. That means that either the author would accidently take the place of ("clobber") a declaration that was meant to be used in the DTD, and thus trigger a cryptic error message, or the DTD designer's entity would clobber the entity that was meant to go in the document instance, and a seemingly random string of DTD-text would appear in the middle of the document! XML prevents this by having two different types of entities with distinct syntaxes for declaration and use.

Parameter entities are distinguished from general entity declarations by a single percent symbol in their declaration, and by a different syntax in their use. Here is an example of a parameter entity declaration and use

Example 33-8. Parameter entity

```
<!DOCTYPE EXAMPLE[
    <!-- parameter entity declaration -->
<!ENTITY % example-entity "<!ELEMENT EXAMPLE (#PCDATA)>">
    <!-- parameter entity use -->
%example-entity;
]>
<EXAMPLE>
</EXAMPLE>
```

The entity in Example 33-8 is declared with a syntax similar to that of general entities, but it has a percent sign between the string `<!ENTITY` and the entity's name. This is what differentiates parameter entity declarations from general entity declarations. If you want a general entity you just leave the percent character out.

The entity contains a complete element type declaration. It is referenced on the line after it is declared. Parameter entity references start with the per-

cent-sign and end with the semicolon. The parser replaces the entity reference with the entity's content. In Example 33-8, the processor replaces the reference with the element type declaration "<!ELEMENT EXAMPLE (#PCDATA)>". It then parses and interprets the element type declaration as if it had occurred there originally. The element type is declared and so the example is valid.

Spec. Reference 33-4. Parameter Entity Declaration

```
[72]  PEDecl ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[74]  PEDef ::=  EntityValue | ExternalID
[75]  ExternalID ::=  'SYSTEM' S SystemLiteral
                    | 'PUBLIC' S PubidLiteral S SystemLiteral
[69]  PEReference ::=  '%' Name ';'
```

Parameter entities can be external, just as general entities can be. But they can never be unparsed. Parameter entities exist to provide building blocks for reusing markup declarations and making DTDs more flexible. It would not make sense to tell the XML processor not to process one! An example of an external parameter entity is in Example 33-9.

Example 33-9. External parameter entity

```
<!DOCTYPE EXAMPLE[
    <!-- parameter entity declaration -->
<!ENTITY % example-entity SYSTEM "pictures.ent">
    <!-- parameter entity use -->
%example-entity;
]>
<EXAMPLE>
</EXAMPLE>
```

Parameter entities cannot be referenced in the document instance. In fact, the percent character is not special in the document instance, so if you try to reference a parameter entity in the instance, you will just get the entity reference text in your data, like "%this;".

Parameter entities can only be referenced after they have been declared. General entities, in contrast, may be referenced before they are declared:

This works because the entity replacement for `&usee;` does not take place until the point where the `user` entity is *referenced.* Remember that general entities can only be expanded in the document instance. So the fact

Example 33-10. General entity usage

```
<!ENTITY user "This entity uses &usee;.">
<!ENTITY usee "<em>another entity</em>">
```

that `user` refers to *usee* is recorded, but the replacement is not immediately done. Later, in the document instance, the author will refer to the `user` entity using the general entity reference, `&user;`. At that point, the inclusion of its replacement text will trigger the expansion of the `&usee;` entity reference and the inclusion of its replacement text.

As you know, all entity declarations are in the DTD. The document instance comes after the DTD. The general entity expansions do not take place until they are referenced in the document instance, so general entity reference expansions will always take place after all of the declarations have been processed, no matter what the order of the general entity declarations in the DTD. Hence, the content of general entities can contain references to other general entities that are declared after them, but the content of parameter entities cannot.

33.8 | Markup may not span entity boundaries

Parsed entities may contain markup as well as character data, but elements and other markup must not span entity boundaries. This means that a particular element may not start in one entity and end in another. If you think of entities as boxes, then an element cannot be half in one box and half in another. This is an example of illegal entity use:

Example 33-11. Elements spanning entity boundaries.

```
<!DOCTYPE EXAMPLE[
    <!ENTITY start "<title>This is a">
    <!ENTITY finish "title</title>">
]>

&start;&finish;
```

This document is not well-formed. When the entity references are replaced with their text, they create a title element. This element spans the entities.

Other markup cannot span entities either. Declarations, comments, processing instructions and entity references must all finish in the entity in which they started. This applies to the document entity as much as any other. Markup strings and elements may not start in the document entity and finish in an included entity. This is a subtle but important rule. Documents which fail to conform are not well-formed.

In Example 33-12, entities are used in ways that are illegal. They are all illegal because they start markup without finishing it or finish it without starting it.

Example 33-12. Illegal entities

```
<!DOCTYPE TEST[
    <!ENTITY illegal1 "This will soon be <em>illegal">
    <!ENTITY illegal2 "This will too <em">
    <!ENTITY illegal3 "This will also </em>">
    <!ENTITY illegal4 "And so will <!-- this">
    <!ENTITY illegal5 "And this &too">
    <!-- note that none of these are illegal yet. -->
...
]><TEST>
<!-- These references are all illegal -->
&illegal1; <!-- Start-tag in entity with no end-tag there. -->
&illegal2; <!-- Start of tag in entity -->
&illegal3; <!-- End-tag in entity with no start-tag there. -->
&illegal4; <!-- Comment start but no end in entity. -->
&illegal5; <!-- Entity reference starts in entity. -->
</TEST>
```

The entities in Example 33-13 can be used legally or illegally. They do not necessarily represent the start or end of elements or markup, because they do not contain the strings that are used to start a tag ("<"), comment ("<!--"), general entity reference ("&") or other markup. Entity content is interpreted as markup if the replacement text would be interpreted as markup in the same context. In other words, the processor expands the entity and then looks for markup. If the markup it finds spans entity boundaries, then it is illegal.

In this case, it is not the declared entities themselves that are causing the problem, but the fact that elements, entities and markup started in the doc-

Example 33-13. Sometimes legal entities

```
<?xml version="1.0"?>
<!DOCTYPE TEST[
<!ELEMENT TEST (#PCDATA)>
<!ENTITY maybelegal1 "em>"> <!-- May not be part of tag -->
<!ENTITY maybelegal2 "-->"> <!-- May not be part of comment -->
<!ENTITY maybelegal3 "ph>"> <!-- May not be part of tag -->
]>
<TEST>
&maybelegal1; <!-- Legal: Interpreted as character data -->
&maybelegal2; <!-- Legal: Interpreted as character data -->
&maybelegal3; <!-- Legal: Interpreted as character data -->

<&maybelegal1; <!-- Illegal: Markup (tag) spans entities -->
<!-- &maybelegal2; <!-- Ignored: entity ref ignored in comment -->
<em&maybelegal3;   <!-- Illegal: Markup (tag) spans entities -->
</TEST>
```

ument entity must end there, just as in any other entity. The context of an entity reference is very important. That is what decides whether it is legal or illegal.

This is true even of entities that hold *complete* tags, elements, comments, processing instructions, character references, or entity references. References to those entities are legal anywhere their replacement text would be legal. The same applies to *validity* (conformance to a document type definition). Example 33-14 is well-formed, but not valid, because the fully expanded document would not be valid. Validity is covered in Chapter 32, "Creating a document type definition ", on page 448.

Example 33-14. Well-formed but not valid

```
<?xml version="1.0"?>
<!DOCTYPE TEST[
  <!ELEMENT EVENT (TIME, DESCRIPTION)>
  <!ELEMENT TIME (#PCDATA)>
  <!ELEMENT DESCRIPTION (#PCDATA)>
  <!ENTITY accident "<ERROR>Error</ERROR>">
]>
<EVENT>&accident;</EVENT>
```

The document in the example is well-formed. Both the EVENT and ERROR elements start and end in the same entity. It meets all of the other rules required for it to be well-formed. But it is not valid, because accident's

replacement text consists of an ERROR element which is not valid where the entity is referenced. (in the EVENT element).

Conceptually, validation occurs after all entities have been parsed.

Spec. Reference 33-5. General entity definition

```
[70]  EntityDecl ::=  GEDecl | PEDecl
[71]  GEDecl ::=  '<!ENTITY' S Name S EntityDef S? '>'
[73]  EntityDef ::=  EntityValue | (ExternalID NDataDecl?)
[72]  PEDecl ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[74]  PEDef ::=  EntityValue | ExternalID
```

33.8.1 *Legal parameter entity reference*

Neither general entities nor parameter entities may span markup boundaries, but parameter entities have other restrictions on them. There are precise places that parameter entity references are allowed. Within the internal subset, the rules are simple: parameter entities can only be expanded in places where full markup declarations are allowed. For them to be legal in these contexts they must always contain one or more markup declarations.

Example 33-15. Multiple markup declarations in one parameter entity

```
<!ENTITY % several-declarations
                "<!ELEMENT FOO (#PCDATA)>
                <!ELEMENT BAR (#PCDATA)>
                <!ELEMENT BAZ (#PCDATA)>"
%several-declarations;
```

Because of the way XML handles white space, this entity declaration's replacement text is parsed as it would if the entity declaration had occurred on a single line. In this case we have defined the literal entity value over several lines to make the DTD more readable. When we refer to the parameter entity "several-declarations", the three element types are declared.

The rules for parameter entities in the external subset are much more complex. This is because parameter entities in the external subset are not restricted to complete markup declarations. They can also be parts of a markup declaration. XML restricts parameter entities in the internal subset to full declarations because the internal subset is supposed to be very easy to process quickly by browsers and other processors. The external subset

allows more complex, powerful parameter entity references. For instance, in the external subset, this would be a legal series of declarations:

Example 33-16. Entities in the external subset

```
<!ENTITY ent-name "the-entity">
<!ENTITY ent-value "This is the entity">
<!ENTITY %ent-name; %ent-value;>
```

Both the name and the replacement text of the final entity declaration are specified through parameter entity references. Their replacement texts become the entity's name and replacement text.

The tricky part is that there are only particular places that you can use parameter entity references in markup declarations. You might wonder, for instance, if you could replace the string "<!ENTITY" with a parameter entity reference. You might guess that this is impossible because XML does not allow a markup declaration to start in one entity and end in another. You would guess correctly. It would be harder to guess whether you could use an entity reference to fill in the string "ENTITY" which follows the "<!" It turns out that this is illegal as well.

To be safe, we would advise you to stick to using parameter entities only to hold full markup declarations until you are familiar with the text of the XML specification itself. The specification uses a special convention in the grammar to describe the places that parameter entity replacement is allowed in the external subset. There are just too many places for us to list them here.

33.9 | External identifiers

External identifiers refer to information outside the entity in which they occur. There are two types. System identifiers use URIs to refer to an object based on its location. Public identifiers use a publicly declared name to refer to information.

Spec. Reference 33-6. External identifier

```
[75]  ExternalID ::=  'SYSTEM' S SystemLiteral
                    | 'PUBLIC' S PubidLiteral S SystemLiteral
```

33.9.1 *System identifiers*

The *SystemLiteral* that follows the keyword SYSTEM is just a URI. Here is another example of that:

```
<!ENTITY ent SYSTEM "http://www.entities.com/ent.xml">
```

You can also use relative URIs to refer to entities on the same machine as the referring entity. A relative URI is one that does not contain a complete machine name and path. The machine name and part of the path are implied from the context.

Example 33-17. Local external general entity

```
<!ENTITY local SYSTEM "local.xml">
```

If this were declared in a document at the URI `http://www.baz.org/`, then the processor would fetch the replacement text from `http://www.baz.org/local.xml`.

These URIs are relative to the location of the referring entity (such as an external parameter entity or the external subset of the DTD) and not necessarily to the document entity. If your document entity is on one machine, and it includes some markup declarations from another machine, relative URIs in the included declarations are interpreted as being on the second machine.

For example, your document might be at `http://www.myhome.com`. It might include a DTD component with a set of pictures of playing cards from `http://www.poker.com/cards.dtd`. If that DTD component had a URI, `4Heartss.gif`, it would be interpreted relative to the poker site, not yours.

33.9.2 *Public identifiers*

It is also possible to refer to a DTD component or any entity by a name, in addition to a URI. This name is called a "public identifier". If a few entities become widely used in XML circles then it would be inefficient for everyone to fetch the entities from the same servers. Instead, their software should come with those entities already installed (or else it should know the most efficient site from which to download them, perhaps from a corporate

intranet). To enable these smarter lookup mechanisms, you would refer to those DTDs by public identifiers, like this:

Example 33-18. Referencing a DTD by public identifier

```
<!DOCTYPE MEMO PUBLIC "-//SGMLSOURCE//DTD MEMO//EN"
                      "http://www.sgmlsource.com/dtds/memo.dtd">
<MEMO> </MEMO>
```

The public identifier is a unique name for the entity. It should be unique world-wide. Usually they contain corporate or personal names to make them more likely to be unique. If the software knows how to translate the public identifier into a URI, it will do so. If not, it will use the system identifier.

Right now, the translation from public identifier to URI is typically either hard-wired into a processor or controlled through files called "entity catalogs". Entity catalogs list public identifiers and describe their URIs, in the same way that phone books allow you to look up a name and find a number. Documentation for XML software should mention the format of the catalogs it supports, if any.

In the future there may be intranet- and Internet-wide systems that will look up a public identifier and download the DTD from the site that is closest to you. The Web's designers have been promising this feature for years and XML is ready when they deliver. In the meantime, the system identifier following the public identifier will be used.

33.10 | Conclusion

As you can see, XML separates issues of logical structure from those of the physical storage of the document. This means that document type designers do not have to forsee every possible reasonable way of breaking up a document when they design the document type. This is good, because that sort of decision is best made by those who know their system resource limits, bandwidth limits, editor preferences, and so forth. The document type designer, in contrast, takes responsibility for deciding on a good structure for the document.