CHAPTER 9

Entities and External DTD Subsets

In This Chapter

What is an entity?

Internal general entities

External general entities

Internal parameter entities

External parameter entities

How to build a document from pieces

Entities and DTDs in well-formed documents

A single XML document may draw both data and declarations from many different sources, in many different files. In fact, some of the data may draw directly from databases, CGI scripts, or other non-file sources. The items where the pieces of an XML file are stored, in whatever form they take, are called entities. Entity references load these entities into the main XML document. General entity references load data into the root element of an XML document, while parameter entity references load data into the document's DTD.

What Is an Entity?

Logically speaking, an XML document is composed of a prolog followed by a root element which strictly contains all other elements. But in practice, the actual data of an XML document can spread across multiple files. For example, each `PLAYER` element might appear in a separate file even though the root element contains all 900 or so players in a league. The storage units that contain particular parts of an XML document are called *entities*. An entity may consist of a file, a database record, or any other item that contains data. For example, all the complete XML files and style sheets in this book are entities.

The storage unit that contains the XML declaration, the document type declaration, and the root element is called the *document entity*. However, the root element and its descendents may also contain entity references pointing to additional data that should be inserted into the document. A validating XML processor combines all the different referenced entities into a single logical document before it passes the document onto the end application or displays the file.

Non-validating processors may, but do not have to, insert external entities. They must insert internal entities.

The primary purpose of an entity is to hold content: well-formed XML, other forms of text, or binary data. The prolog and the document type declaration are part of the root entity of the document they belong to. An XSL style sheet qualifies as an entity, but only because it itself is a well-formed XML document. The entity that makes up the style sheet is not one of the entities that composes the XML document to which the style sheet applies. A CSS style sheet is not an entity at all.

Most entities have names by which you can refer to them. The only exception is the document entity-the main file containing the XML document (although there's no requirement that this be a file as opposed to a database record, the output of a CGI program, or something else). The document entity is the storage unit, in whatever form it takes, that holds the XML declaration, the document type declaration (if any), and the root element. Thus, every XML document has at least one entity.

There are two kinds of entities: internal and external. Internal entities are defined completely within the document entity. The document itself is one such entity, so all XML documents have at least one internal entity.

External entities, by contrast, draw their content from another source located via a URL. The main document only includes a reference to the URL where the actual content resides. In HTML, an `IMG` element represents an external entity (the actual image data) while the document itself contained between the `<HTML>` and `</HTML>` tags is an internal entity.

Entities fall into two categories: parsed and unparsed. Parsed entities contain well-formed XML text. Unparsed entities contain either binary data or non-XML text (like an email message). Currently, unparsed entities aren't well supported (if at all) by most XML processors. In this chapter, we focus on parsed entities.

Chapter 11, *Embedding Non-XML Data*, covers unparsed entities.

Entity references enable data from multiple entities to merge together to form a single document. General entity references merge data into the document content. Parameter entity references merge declarations into the document's DTD. `<`, `>`, `'`, `"e;`, and `&` are predefined entity references that refer to the text entities `<`, `>`, `'`, `"`, and `&`, respectively. However, you can also define new entities in your document's DTD.

Internal General Entities

You can think of an internal general entity reference as an abbreviation for commonly used text or text that's hard to type. An <!ENTITY> tag in the DTD defines an abbreviation and the text the abbreviation stands for. For instance, instead of typing the same footer at the bottom of every page, you can simply define that text as the footer entity in the DTD and then type &footer; at the bottom of each page. Furthermore, if you decide to change the footer block (perhaps because your email address changes), you only need to make the change once in the DTD instead of on every page that shares the footer.

General entity references begin with an ampersand (&) and end with a semicolon (;), with the entity's name between these two characters. For instance, < is a general entity reference for the less than sign (<) The name of this entity is lt. The replacement text of this entity is the one character string <. Entity names consist of any set of alphanumeric characters and the underscore. Whitespace and other punctuation characters are prohibited. Like most everything else in XML, entity references are case sensitive.

Although the colon (:) is technically permitted in entity names, this character is reserved for use with namespaces, which are discussed in Chapter 18.

Defining an Internal General Entity Reference

Internal general entity references are defined in the DTD with the <!ENTITY> tag, which has the following format:

```
<!ENTITY name "replacement text">
```

The *name* is the abbreviation for the *replacement text*. The replacement text must be enclosed in quotation marks because it may contain whitespace and XML markup. You type the name of the entity in the document, but the reader sees the replacement text.

For example, my name is the somewhat excessive "Elliotte Rusty Harold" (blame my parents for that one). Even with years of practice, I still make typos with that phrase. I can define a general entity reference for my name so that every time I type &ERH;, the reader will see "Elliotte Rusty Harold". That definition is:

```
<!ENTITY ERH "Elliotte Rusty Harold">
```

Listing 9-1 demonstrates the &ERH; general entity reference. Figure 9-1 shows this document loaded into Internet Explorer. You see that the &ERH; entity reference in the source code is replaced by Elliotte Rusty Harold in the output.

Listing 9-1: The ERH internal general entity reference

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

   <!ENTITY ERH "Elliotte Rusty Harold">

   <!ELEMENT DOCUMENT (TITLE, SIGNATURE)>
   <!ELEMENT TITLE (#PCDATA)>
   <!ELEMENT COPYRIGHT (#PCDATA)>
   <!ELEMENT EMAIL (#PCDATA)>
   <!ELEMENT LAST_MODIFIED (#PCDATA)>
   <!ELEMENT SIGNATURE (COPYRIGHT, EMAIL, LAST_MODIFIED)>
]>
<DOCUMENT>
  <TITLE>&ERH;</TITLE>
  <SIGNATURE>
    <COPYRIGHT>1999 &ERH;</COPYRIGHT>
    <EMAIL>elharo@metalab.unc.edu</EMAIL>
    <LAST_MODIFIED>March 10, 1999</LAST_MODIFIED>
   </SIGNATURE>
</DOCUMENT>
```

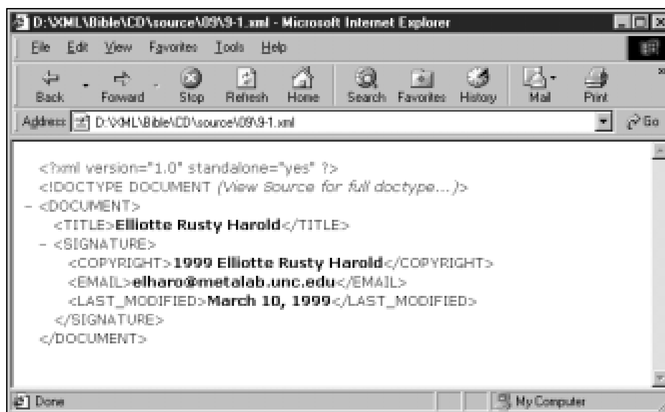


Figure 9-1: Listing 9-1 after the internal general entity reference has been replaced by the actual entity

Notice that the general entity reference, `&ERH;` appears inside both the `COPYRIGHT` and `TITLE` elements even though these are declared to accept only `#PCDATA` as children. This arrangement is legal because the replacement text of the `&ERH;` entity reference is parsed character data. Validation is done against the document after all entity references have been replaced by their values.

The same thing occurs when you use a style sheet. The styles are applied to the element tree as it exists after entity values replace the entity references.

You can follow the same model to declare general entity references for the copyright, the email address, or the last modified date:

```
<!ENTITY COPY99 "Copyright 1999">
<!ENTITY EMAIL "elharo@metalab.unc.edu">
<!ENTITY LM "Last modified: ">
```

I omitted the date in the `&LM;` entity because it's likely to change from document to document. There is no advantage to making it an entity reference.

Now you can rewrite the document part of Listing 9-1 even more compactly:

```
<DOCUMENT>
  <TITLE>&ERH;</TITLE>
  <SIGNATURE>
    <COPYRIGHT>&COPY99; &ERH;</COPYRIGHT>
    <EMAIL>&EMAIL;</EMAIL>
    <LAST_MODIFIED>&LM; March 10, 1999</LAST_MODIFIED>
   </SIGNATURE>
</DOCUMENT>
```

One of the advantages of using entity references instead of the full text is that these references make it easy to change the text. This is especially useful when a single DTD is shared between multiple documents. (You'll learn this skill in the section on sharing common DTDs among documents.) For example, suppose I decide to use the email address `eharold@solar.stanford.edu` instead of `elharo@metalab.unc.edu`. Rather than searching and replacing through multiple files, I simply change one line of the DTD as follows:

```
<!ENTITY EMAIL "eharold@solar.stanford.edu">
```

Using General Entity References in the DTD

You may wonder whether it's possible to include one general entity reference inside another as follows:

```
<!ENTITY COPY99 "Copyright 1999 &ERH;">
```

This example is in fact valid, because the `ERH` entity appears as part of the `COPY99` entity that itself will ultimately become part of the document's content. You can also use general entity references in other places in the DTD that ultimately become part of the document content (such as a default attribute value), although there are restrictions. The first restriction: The statement cannot use a circular reference like this one:

```
<!ENTITY ERH "&COPY99 Elliotte Rusty Harold">
<!ENTITY COPY99 "Copyright 1999 &ERH;">
```

The second restriction: General entity references may not insert text that is only part of the DTD and will not be used as part of the document content. For example, the following attempted shortcut fails:

```
<!ENTITY  PCD    "(#PCDATA)">
<!ELEMENT ANIMAL &PCD;>
<!ELEMENT FOOD   &PCD;>
```

It's often useful, however, to have entity references merge text into a document's DTD. For this purpose, XML uses the parameter entity reference, which is discussed later in this chapter.

The only restriction on general entity values is that they may not contain the three characters `%`, `&`, and `"` directly, though you can include them via character references. `&` and `%` may be included if they're starting an entity reference rather than simply representing themselves. The lack of restrictions means that an entity may contain tags and span multiple lines. For example, the following `SIGNATURE` entity is valid:

```
<!ENTITY SIGNATURE
  "<SIGNATURE>
     <COPYRIGHT>1999 Elliotte Rusty Harold</COPYRIGHT>
     <EMAIL>elharo@metalab.unc.edu</EMAIL>
     <LAST_MODIFIED>March 10, 1999</LAST_MODIFIED>
   </SIGNATURE>"
>
```

The next obvious question is whether it's possible for entities to have parameters. Can you use the above `SIGNATURE` entity but change the date in each separate `LAST_MODIFIED` element on each page? The answer is no; entities are only for static replacement text. If you need to pass data to an entity, you should use a tag along with the appropriate rendering instructions in the style sheet instead.

Predefined General Entity References

XML predefines five general entity references, as listed in Table 9-1. These five entity references appear in XML documents in place of specific characters that would otherwise be interpreted as markup. For instance, the entity reference `<` stands for the less-than sign (`<`), which could be interpreted as the beginning of a tag.

For maximum compatibility, you should declare these references in your DTD if you plan to use them. Declaration is actually quite tricky because you must also escape the characters in the DTD without using recursion. To declare these references, use character references containing the hexadecimal ASCII value of each character. Listing 9-2 shows the necessary declarations:

Table 9-1
XML Predefined Entity References

Entity Reference	*Character*
`&`	&
`<`	<
`>`	>
`"`	"
`'`	'

Listing 9-2: **Declarations for predefined general entity references**

```
<!ENTITY lt   "&#38;#60;">
<!ENTITY gt   "&#62;">
<!ENTITY amp  "&#38;#38;">
<!ENTITY apos "&#39;">
<!ENTITY quot "&#34;">
```

External General Entities

External entities are data outside the main file containing the root element/document entity. External entity references let you embed these external entities in your document and build XML documents from multiple independent files.

Documents using only internal entities closely resemble the HTML model. The complete text of the document is available in a single file. Images, applets, sounds, and other non-HTML data may be linked in, but at least all the text is present. Of course, the HTML model has some problems. In particular, it's quite difficult to embed dynamic information in the file. You can embed dynamic information by using CGI, Java applets, fancy database software, server side includes, and various other means, but HTML alone only provides a static document. You have to go outside HTML to build a document from multiple pieces. Frames are perhaps the simplest HTML solution to this problem, but they are a user interface disaster that consistently confuse and annoy users.

Part of the problem is that one HTML document does not naturally fit inside another. Every HTML document should have exactly one `BODY`, but no more. Server side includes only enable you to embed fragments of HTML—never an entire valid document—inside a document. In addition, server side includes are server dependent and not truly part of HTML.

XML, however, is more flexible. One document's root element is not necessarily the same as another document's root element. Even if two documents share the same root element, the DTD may declare that the element is allowed to contain itself. The XML standard does not prevent well-formed XML documents from being embedded in other well-formed XML documents when convenient.

XML goes further, however, by defining a mechanism whereby an XML document can be built out of multiple smaller XML documents found either on local or remote systems. The parseris responsible for merging all the different documents together in a fixed order. Documents may contain other documents, which may contain other documents. As long as there's no recursion (an error reported by the processor), the application only sees a single, complete document. In essence, this provides client-side includes.

With XML, you can use an external general entity reference to embed one document in another. In the DTD, you declare the external reference with the following syntax:

```
<!ENTITY name SYSTEM "URI">
```

Note

URI stands for Uniform Resource Identifier. URIs are similar to URLs but allow for more precise specification of the linked resource. In theory, URIs separate the resource from the location so a Web browser can select the nearest or least congested of several mirrors without requiring an explicit link to that mirror. URIs are an area of active research and heated debate. Therefore, in practice and certainly in this book, URIs are URLs for all purposes.

For example, you may want to put the same signature block on almost every page of a site. For the sake of definiteness, let's assume the signature block is the XML code shown in Listing 9-3. Furthermore, let's assume that you can retrieve this code from the URL `http://metalab.unc.edu/xml/signature.xml`.

Listing 9-3: **An XML signature file**

```
<?xml version="1.0"?>
<SIGNATURE>
  <COPYRIGHT>1999 Elliotte Rusty Harold</COPYRIGHT>
  <EMAIL>elharo@metalab.unc.edu</EMAIL>
</SIGNATURE>
```

Associate this file with the entity reference `&SIG;` by adding the following declaration to the DTD:

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

You can also use a relative URL. For example,

```
<!ENTITY SIG SYSTEM "/xml/signature.xml">
```

If the file to be included is in the same directory as the file doing the including, you only need to use the file name. For example,

```
<!ENTITY SIG SYSTEM "signature.xml">
```

With any of these declarations, you can include the contents of the signature file in a document at any point merely by using `&SIG;`, as illustrated with the simple document in Listing 9-4. Figure 9-2 shows the rendered document in Internet Explorer 5.0.

Listing 9-4: The SIG external general entity reference

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [
   <!ELEMENT DOCUMENT (TITLE, SIGNATURE)>
   <!ELEMENT TITLE (#PCDATA)>
   <!ELEMENT COPYRIGHT (#PCDATA)>
   <!ELEMENT EMAIL (#PCDATA)>
   <!ELEMENT SIGNATURE (COPYRIGHT, EMAIL)>
   <!ENTITY SIG SYSTEM
      "http://metalab.unc.edu/xml/signature.xml">
]>
<DOCUMENT>
  <TITLE>Entity references</TITLE>
  &SIG;
</DOCUMENT>
```

Aside from the addition of the external entity reference, note that the `standalone` attribute of the XML declaration now has the value `no` because this file is no longer complete. Parsing the file requires additional data from the external file signature.xml.

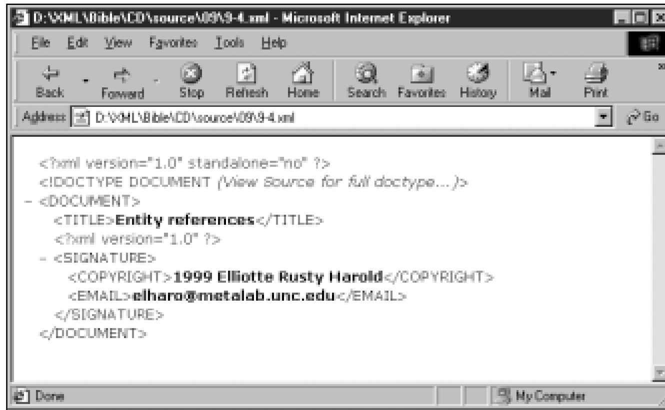


Figure 9-2: A document that uses an external general entity reference.

Internal Parameter Entities

General entities become part of the document, not the DTD. They can be used in the DTD but only in places where they will become part of the document body. General entity references may not insert text that is only part of the DTD and will not be used as part of the document content. It's often useful, however, to have entity references in a DTD. For this purpose, XML provides the *parameter entity reference*.

Parameter entity references are very similar to general entity references—with these two key differences:

1. Parameter entity references begin with a percent sign (%) rather than an ampersand (&).
2. Parameter entity references can only appear in the DTD, not the document content.

Parameter entities are declared in the DTD like general entities with the addition of a percent sign before the name. The syntax looks like this:

```
<!ENTITY % name "replacement text">
```

The name is the abbreviation for the entity. The reader sees the replacement text, which must appear in quotes. For example:

```
<!ENTITY % ERH "Elliotte Rusty Harold">
<!ENTITY COPY99 "Copyright 1999 %ERH;">
```

Our earlier failed attempt to abbreviate (#PCDATA) works when a parameter entity reference replaces the general entity reference:

```
<!ENTITY % PCD "(#PCDATA)">
<!ELEMENT ANIMAL %PCD;>
<!ELEMENT FOOD %PCD;>
```

The real value of parameter entity references appears in sharing common lists of children and attributes between elements. The larger the block of text you're replacing and the more times you use it, the more useful parameter entity references become. For instance, suppose your DTD declares a number of block level container elements like PARAGRAPH, CELL, and HEADING. Each of these container elements may contain an indefinite number of inline elements like PERSON, DEGREE, MODEL, PRODUCT, ANIMAL, INGREDIENT, and so forth. The element declarations for the container elements could appear as the following:

```
<!ELEMENT PARAGRAPH
   (PERSON | DEGREE | MODEL | PRODUCT | ANIMAL | INGREDIENT)*>
<!ELEMENT CELL
   (PERSON | DEGREE | MODEL | PRODUCT | ANIMAL | INGREDIENT)*>
<!ELEMENT HEADING
   (PERSON | DEGREE | MODEL | PRODUCT | ANIMAL | INGREDIENT)*>
```

The container elements all have the same contents. If you invent a new element like EQUATION, CD, or ACCOUNT, this element must be declared as a possible child of all three container elements. Adding it to two, but forgetting to add it to the third element, may cause trouble. This problem multiplies when you have 30 or 300 container elements instead of three.

The DTD is much easier to maintain if you don't give each container a separate child list. Instead, make the child list a parameter entity reference; then use that parameter entity reference in each of the container element declarations. For example:

```
<!ENTITY % inlines
  "(PERSON | DEGREE | MODEL | PRODUCT | ANIMAL | INGREDIENT)*">
<!ELEMENT PARAGRAPH %inlines;>
<!ELEMENT CELL %inlines;>
<!ELEMENT HEADING %inlines;>
```

To add a new element, you only have to change a single parameter entity declaration, rather than three, 30, or 300 element declarations.

Parameter entity references must be declared before they're used. The following example is invalid because the %PCD; reference is not declared until it's already been used twice:

```
<!ELEMENT FOOD %PCD;>
<!ELEMENT ANIMAL %PCD;>
<!ENTITY % PCD "(#PCDATA)">
```

Parameter entities can only be used to provide part of a declaration in the external DTD subset. That is, parameter entity references can only appear inside a declaration in the external DTD subset. The above examples are all invalid if they're used in an internal DTD subset.

In the internal DTD subset, parameter entity references can only be used outside of declarations. For example, the following is valid in both the internal and external DTD subsets:

```
<!ENTITY % hr "<!ELEMENT HR EMPTY>">
%hr;
```

Of course, this really isn't any easier than declaring the HR element without parameter entity references:

```
<!ELEMENT HR EMPTY>
```

You'll mainly use parameter entity references in internal DTD subsets when they're referring to external parameter entities; that is, when they're pulling in declarations or parts of declarations from a different file. This is the subject of the next section.

External Parameter Entities

The preceding examples used monolithic DTDs that define all the elements used in the document. This technique becomes unwieldy with longer documents, however. Furthermore, you often want to use part of a DTD in many different places.

For example, consider a DTD that describes a snail mail address. The definition of an address is quite general, and can easily be used in many different contexts. Similarly, the list of predefined entity references in Listing 9-2 is useful in most XML files, but you'd rather not copy and paste it all the time.

External parameter entities enable you to build large DTDs from smaller ones. That is, one external DTD may link to another and in so doing pull in the elements and entities declared in the first. Although cycles are prohibited—DTD 1 may not refer to DTD 2 if DTD 2 refers to DTD 1—such nested DTDs can become large and complex.

At the same time, breaking a DTD into smaller, more manageable chunks makes the DTD easier to analyze. Many of the examples in the last chapter were unnecessarily large because an entire document and its complete DTD were stored in a single file. Both the document and its DTD become much easier to understand when split into separate files.

Furthermore, using smaller, modular DTDs that only describe one set of elements makes it easier to mix and match DTDs created by different people or organizations. For instance, if you're writing a technical article about high temperature superconductivity, you can use a molecular sciences DTD to describe the molecules involved, a math DTD to write down your equations, a vector graphics DTD for the figures, and a basic HTML DTD to handle the explanatory text.

Note

In particular, you can use the mol.dtd DTD from Peter Murray-Rust's Chemical Markup Language, the MathML DTD from the W3C's Mathematical Markup Language, the SVG DTD for the W3C's Scalable Vector Graphics, and the W3C's HTML-in-XML DTD.

You can probably think of more examples where you need to mix and match concepts (and therefore tags) from different fields. Human thought doesn't restrict itself to narrowly defined categories. It tends to wander all over the map. The documents you write will reflect this.

Let's see how to organize the baseball statistics DTD as a combination of several different DTDs. This example is extremely hierarchical. One possible division is to write separate DTDs for `PLAYER`, `TEAM`, and `SEASON`. This is far from the only way to divide the DTD into more manageable chunks, but it will serve as a reasonable example. Listing 9-5 shows a DTD solely for a player that can be stored in a file named player.dtd:

Listing 9-5: A DTD for the PLAYER element and its children (player.dtd)

```
<!— Player Info —>
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, P, G,
  GS, AB?, R?, H?, D?, T?, HR?, RBI?, SB?, CS?,
  SH?, SF?, E?, BB?, S?, HBP?, W?, L?, SV?, CG?, SO?, ERA?,
  IP?, HRA?, RA?, ER?, HB?, WP?, B?, WB?, K?)
>

<!— Player's last name —>
<!ELEMENT SURNAME (#PCDATA)>

<!— Player's first name —>
<!ELEMENT GIVEN_NAME (#PCDATA)>

<!— Position —>
<!ELEMENT P (#PCDATA)>

<!—Games Played —>
<!ELEMENT G (#PCDATA)>
```

Continued

Listing 9-5 *(continued)*

```
  <!--Games Started -->
  <!ELEMENT GS (#PCDATA)>


<!-- ======================= -->
  <!-- Batting Statistics -->
  <!-- At Bats -->
  <!ELEMENT AB (#PCDATA)>

  <!-- Runs -->
  <!ELEMENT R (#PCDATA)>

  <!-- Hits -->
  <!ELEMENT H (#PCDATA)>

  <!-- Doubles -->
  <!ELEMENT D (#PCDATA)>

  <!-- Triples -->
  <!ELEMENT T (#PCDATA)>

  <!-- Home Runs -->
  <!ELEMENT HR (#PCDATA)>

  <!-- Runs Batted In -->
  <!ELEMENT RBI (#PCDATA)>

  <!-- Stolen Bases -->
  <!ELEMENT SB (#PCDATA)>

  <!-- Caught Stealing -->
  <!ELEMENT CS (#PCDATA)>

  <!-- Sacrifice Hits -->
  <!ELEMENT SH (#PCDATA)>

  <!-- Sacrifice Flies -->
  <!ELEMENT SF (#PCDATA)>

  <!-- Errors -->
  <!ELEMENT E (#PCDATA)>

  <!-- Walks (Base on Balls) -->
  <!ELEMENT BB (#PCDATA)>

  <!-- Struck Out -->
  <!ELEMENT S (#PCDATA)>

  <!-- Hit By Pitch -->
  <!ELEMENT HBP (#PCDATA)>
```

```
<!-- ====================== -->
  <!-- Pitching Statistics -->
  <!-- Complete Games -->
  <!ELEMENT CG (#PCDATA)>

  <!-- Wins -->
  <!ELEMENT W (#PCDATA)>

  <!-- Losses -->
  <!ELEMENT L (#PCDATA)>

  <!-- Saves -->
  <!ELEMENT SV (#PCDATA)>

  <!-- Shutouts -->
  <!ELEMENT SO (#PCDATA)>

  <!-- ERA -->
  <!ELEMENT ERA (#PCDATA)>

  <!-- Innings Pitched -->
  <!ELEMENT IP (#PCDATA)>

  <!-- Home Runs hit Against -->
  <!ELEMENT HRA (#PCDATA)>

  <!-- Runs hit Against -->
  <!ELEMENT RA (#PCDATA)>

  <!-- Earned Runs -->
  <!ELEMENT ER (#PCDATA)>

  <!-- Hit Batter -->
  <!ELEMENT HB (#PCDATA)>

  <!-- Wild Pitches -->
  <!ELEMENT WP (#PCDATA)>

  <!-- Balk -->
  <!ELEMENT B (#PCDATA)>

  <!-- Walked Batter -->
  <!ELEMENT WB (#PCDATA)>

  <!-- Struck Out Batter -->
  <!ELEMENT K (#PCDATA)>

<!-- ====================== -->
  <!-- Fielding Statistics -->
  <!-- Not yet supported -->
```

By itself, this DTD doesn't enable you to create very interesting documents. Listing 9-6 shows a simple valid file that only uses the PLAYER DTD in Listing 9-5. By itself, this simple file is not important; however, you can build other, more complicated files out of these small parts.

Listing 9-6: A valid document using the PLAYER DTD

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE PLAYER SYSTEM "player.dtd">
<PLAYER>
  <GIVEN_NAME>Chris</GIVEN_NAME>
  <SURNAME>Hoiles</SURNAME>
  <P>Catcher</P>
  <G>97</G>
  <GS>81</GS>
  <AB>267</AB>
  <R>36</R>
  <H>70</H>
  <D>12</D>
  <T>0</T>
  <HR>15</HR>
  <RBI>56</RBI>
  <SB>0</SB>
  <CS>1</CS>
  <SH>5</SH>
  <SF>4</SF>
  <E>3</E>
  <BB>38</BB>
  <S>50</S>
  <HBP>4</HBP>
</PLAYER>
```

What other parts of the document can have their own DTDs? Obviously, a `TEAM` is a big part. You could write its DTD as follows:

```
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
```

On closer inspection, however, you should notice that something is missing: the definition of the PLAYER element. The definition is in the separate file player.dtd and needs to be connected to this DTD.

You connect DTDs with external parameter entity references. For a private DTD, this connection takes the following form:

```
<!ENTITY % name SYSTEM "URI">
%name;
```

For example:

```
<!ENTITY % player SYSTEM "player.dtd">
%player;
```

This example uses a relative URL (player.dtd) and assumes that the file player.dtd will be found in the same place as the linking DTD. If that's not the case, you can use a full URL as follows:

```
<!ENTITY % player SYSTEM
   "http://metalab.unc.edu/xml/dtds/player.dtd">
%player;
```

Listing 9-7 shows a completed TEAM DTD that includes a reference to the PLAYER DTD:

Listing 9-7: The TEAM DTD (team.dtd)

```
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
<!ENTITY % player SYSTEM "player.dtd">
%player;
```

A SEASON contains LEAGUE, DIVISION, and TEAM elements. Although LEAGUE and DIVISION could each have their own DTD, it doesn't pay to go overboard with splitting DTDs. Unless you expect you'll have some documents that contain LEAGUE or DIVISION elements that are not part of a SEASON, you might as well include all three in the same DTD. Listing 9-8 demonstrates.

Listing 9-8: The SEASON DTD (season.dtd)

```
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT LEAGUE (LEAGUE_NAME, DIVISION, DIVISION, DIVISION)>

<!-- American or National -->
<!ELEMENT LEAGUE_NAME (#PCDATA)>

<!-- East, West, or Central -->
<!ELEMENT DIVISION_NAME (#PCDATA)>
<!ELEMENT DIVISION (DIVISION_NAME, TEAM+)>
<!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
<!ENTITY % team SYSTEM "team.dtd">
%team;
```

Building a Document from Pieces

The baseball examples have been quite large. Although only a truncated version with limited numbers of players appears in this book, the full document is more than half a megabyte, way too large to comfortably download or search, especially if the reader is only interested in a single team, player, or division. The techniques discussed in the previous section of this chapter allow you to split the document into many different, smaller, more manageable documents, one for each team, player, division, and league. External entity references connect the players to form teams, the teams to form divisions, the divisions to form leagues, and the leagues to form a season.

Unfortunately you cannot embed just any XML document as an external parsed entity. Consider, for example, Listing 9-9, ChrisHoiles.xml. This is a revised version of Listing 9-6. However, if you look closely you'll notice that the prolog is different. Listing 9-6's prolog is:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE PLAYER SYSTEM "player.dtd">
```

Listing 9-9's prolog is simply the XML declaration with no `standalone` attribute and with an `encoding` attribute. Furthermore the document type declaration is completely omitted. In a file like Listing 9-9 that's meant to be embedded in another document, this sort of XML declaration is called a *text declaration*, though as you can see it's really just a legal XML declaration.

Listing 9-9: ChrisHoiles.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<PLAYER>
  <GIVEN_NAME>Chris</GIVEN_NAME>
  <SURNAME>Hoiles</SURNAME>
  <P>Catcher</P>
  <G>97</G>
  <GS>81</GS>
  <AB>267</AB>
  <R>36</R>
  <H>70</H>
  <D>12</D>
  <T>0</T>
  <HR>15</HR>
  <RBI>56</RBI>
  <SB>0</SB>
  <CS>1</CS>
  <SH>5</SH>
  <SF>4</SF>
  <E>3</E>
  <BB>38</BB>
  <S>50</S>
  <HBP>4</HBP>
</PLAYER>
```

I'll spare you the other 1,200 or so players, although you'll find them all on the accompanying CD-ROM in the examples/baseball/players folder.

Text declarations must have an `encoding` attribute (unlike XML declarations which may but do not have to have an `encoding` attribute) that specifies the character set the entity uses. This allows compound documents to be assembled from entities written in different character sets. For example, a document in Latin-5 might combine with a document in UTF-8. The processor/browser still has to understand all the encodings used by the different entities.

The examples in this chapter are all given in ASCII. Since ASCII is a strict subset of both ISO Latin-1 and UTF-8, you could use either of these text declarations:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml version="1.0" encoding="UTF-8"?>
```

Listing 9-10, mets.dtd, and Listing 9-11, mets.xml, show how you can use external parsed entities to put together a complete team. The DTD defines external entity references for each player on the team. The XML document loads the DTD using an

external parameter entity reference in its internal DTD subset. Then, its document entity includes many external general entity references that load in the individual players.

Listing 9-10: The New York Mets DTD with entity references for players (mets.dtd)

```
<!ENTITY AlLeiter SYSTEM "mets/AlLeiter.xml">
<!ENTITY ArmandoReynoso SYSTEM "mets/ArmandoReynoso.xml">
<!ENTITY BobbyJones SYSTEM "mets/BobbyJones.xml">
<!ENTITY BradClontz SYSTEM "mets/BradClontz.xml">
<!ENTITY DennisCook SYSTEM "mets/DennisCook.xml">
<!ENTITY GregMcmichael SYSTEM "mets/GregMcmichael.xml">
<!ENTITY HideoNomo SYSTEM "mets/HideoNomo.xml">
<!ENTITY JohnFranco SYSTEM "mets/JohnFranco.xml">
<!ENTITY JosiasManzanillo SYSTEM "mets/JosiasManzanillo.xml">
<!ENTITY OctavioDotel SYSTEM "mets/OctavioDotel.xml">
<!ENTITY RickReed SYSTEM "mets/RickReed.xml">
<!ENTITY RigoBeltran SYSTEM "mets/RigoBeltran.xml">
<!ENTITY WillieBlair SYSTEM "mets/WillieBlair.xml">
```

Figure 9-3 shows the XML document loaded into Internet Explorer. Notice that all data for all players is present even though the main document only contains references to the entities where the player data resides. Internet Explorer resolves external references-not all XML parsers/browsers do.

You can find the remaining teams on the CD-ROM in the directory examples/baseball. Notice in particular how compactly external entity references enable you to embed multiple players.

Listing 9-11: The New York Mets with players loaded from external entities (mets.xml)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE TEAM SYSTEM "team.dtd" [
  <!ENTITY % players SYSTEM "mets.dtd">
  %players;
]>
<TEAM>
  <TEAM_CITY>New York</TEAM_CITY>
  <TEAM_NAME>Mets</TEAM_NAME>
  &AlLeiter;
```

```
    &ArmandoReynoso;
    &BobbyJones;
    &BradClontz;
    &DennisCook;
    &GregMcmichael;
    &HideoNomo;
    &JohnFranco;
    &JosiasManzanillo;
    &OctavioDotel;
    &RickReed;
    &RigoBeltran;
    &WillieBlair;
  </TEAM>
```

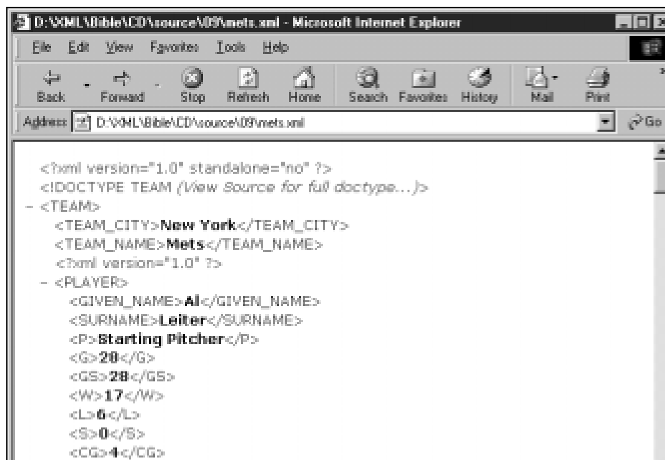


Figure 9-3: The XML document displays all players on the 1998 New York Mets.

It would be nice to continue this procedure building a division by combining team files, a league by combining divisions, and a season by combining leagues. Unfortunately, if you try this you rapidly run into a wall. The documents embedded via external entities cannot have their own DTDs. At most, their prolog can contain the text declaration. This means you can only have a single level of document embedding. This contrasts with DTD embedding where DTDs can be nested arbitrarily deeply.

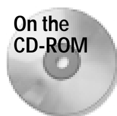
Thus, your only likely alternative is to include all teams, divisions, leagues, and seasons in a single document which refers to the many different player documents. This requires a few more than 1,200 entity declarations (one for each player). Since DTDs can nest arbitrarily, we begin with a DTD that pulls in DTDs like Listing 9-10 containing entity definitions for all the teams. This is shown in Listing 9-12:

Listing 9-12: The players DTD (players.dtd)

```
<!ENTITY % angels SYSTEM "angels.dtd">
%angels;
<!ENTITY % astros SYSTEM "astros.dtd">
%astros;
<!ENTITY % athletics SYSTEM "athletics.dtd">
%athletics;
<!ENTITY % bluejays SYSTEM "bluejays.dtd">
%bluejays;
<!ENTITY % braves SYSTEM "braves.dtd">
%braves;
<!ENTITY % brewers SYSTEM "brewers.dtd">
%brewers;
<!ENTITY % cubs SYSTEM "cubs.dtd">
%cubs;
<!ENTITY % devilrays SYSTEM "devilrays.dtd">
%devilrays;
<!ENTITY % diamondbacks SYSTEM "diamondbacks.dtd">
%diamondbacks;
<!ENTITY % dodgers SYSTEM "dodgers.dtd">
%dodgers;
<!ENTITY % expos SYSTEM "expos.dtd">
%expos;
<!ENTITY % giants SYSTEM "giants.dtd">
%giants;
<!ENTITY % indians SYSTEM "indians.dtd">
%indians;
<!ENTITY % mariners SYSTEM "mariners.dtd">
%mariners;
<!ENTITY % marlins SYSTEM "marlins.dtd">
%marlins;
<!ENTITY % mets SYSTEM "mets.dtd">
%mets;
<!ENTITY % orioles SYSTEM "orioles.dtd">
%orioles;
<!ENTITY % padres SYSTEM "padres.dtd">
%padres;
<!ENTITY % phillies SYSTEM "phillies.dtd">
%phillies;
<!ENTITY % pirates SYSTEM "pirates.dtd">
%pirates;
<!ENTITY % rangers SYSTEM "rangers.dtd">
%rangers;
<!ENTITY % redsox SYSTEM "redsox.dtd">
%redsox;
<!ENTITY % reds SYSTEM "reds.dtd">
%reds;
<!ENTITY % rockies SYSTEM "rockies.dtd">
%rockies;
```

```
<!ENTITY % royals SYSTEM "royals.dtd">
%royals;
<!ENTITY % tigers SYSTEM "tigers.dtd">
%tigers;
<!ENTITY % twins SYSTEM "twins.dtd">
%twins;
<!ENTITY % whitesox SYSTEM "whitesox.dtd">
%whitesox;
<!ENTITY % yankees SYSTEM "yankees.dtd">
%yankees;
```

Listing 9-13, a master document, pulls together all the player sub-documents as well as the DTDs that define the entities for each player. Although this document is much smaller than the monolithic document developed earlier (32K vs. 628K), it's still quite long, so not all players are included here. The full version of Listing 9-13 relies on 33 DTDs and over 1,000 XML files to produce the finished document. The largest problem with this approach is that it requires over 1000 separate connections to the Web server before the document can be displayed.



The full example is on the CD-ROM in the file examples/baseball/players/index.xml.

Listing 9-13: Master document for the 1998 season using external entity references for players

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SEASON SYSTEM "baseball.dtd" [

   <!ENTITY % players SYSTEM "players.dtd">
   %players;

]>
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>National</LEAGUE_NAME>
    <DIVISION>
       <DIVISION_NAME>East</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Florida</TEAM_CITY>
          <TEAM_NAME>Marlins</TEAM_NAME>
        </TEAM>
        <TEAM>
          <TEAM_CITY>Montreal</TEAM_CITY>
          <TEAM_NAME>Expos</TEAM_NAME>
```

Continued

Listing 9-13 *(continued)*

```
        </TEAM>
        <TEAM>
          <TEAM_CITY>New York</TEAM_CITY>
          <TEAM_NAME>Mets</TEAM_NAME>
            &RigoBeltran;
            &DennisCook;
            &SteveDecker;
            &JohnFranco;
            &MattFranco;
            &ButchHuskey;
            &BobbyJones;
            &MikeKinkade;
            &HideoNomo;
            &VanceWilson;
        </TEAM>
        <TEAM>
          <TEAM_CITY>Philadelphia</TEAM_CITY>
          <TEAM_NAME>Phillies</TEAM_NAME>
        </TEAM>
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>Central</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Chicago</TEAM_CITY>
          <TEAM_NAME>Cubs</TEAM_NAME>
        </TEAM>
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>West</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Arizona</TEAM_CITY>
          <TEAM_NAME>Diamondbacks</TEAM_NAME>
        </TEAM>
    </DIVISION>
  </LEAGUE>
  <LEAGUE>
    <LEAGUE_NAME>American</LEAGUE_NAME>
    <DIVISION>
       <DIVISION_NAME>East</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Baltimore</TEAM_CITY>
          <TEAM_NAME>Orioles</TEAM_NAME>
        </TEAM>
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>Central</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Chicago</TEAM_CITY>
```

```
          <TEAM_NAME>White Sox</TEAM_NAME>
            &JeffAbbott;
            &MikeCameron;
            &MikeCaruso;
            &LarryCasian;
            &TomFordham;
            &MarkJohnson;
            &RobertMachado;
            &JimParque;
            &ToddRizzo;
        </TEAM>
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>West</DIVISION_NAME>
        <TEAM>
          <TEAM_CITY>Anaheim</TEAM_CITY>
          <TEAM_NAME>Angels</TEAM_NAME>
        </TEAM>
     </DIVISION>
  </LEAGUE>
</SEASON>
```

You do have some flexibility in which levels you choose for your master document and embedded data. For instance, one alternative to the structure used by Listing 9-12 places the teams and all their players in individual documents, then combines those team files into a season with external entities as shown in Listing 9-14. This has the advantage of using a smaller number of XML files of more even sizes that places less load on the Web server and would download and display more quickly. To be honest, however, the intrinsic advantage of one approach or the other is minimal. Feel free to use whichever one more closely matches the organization of your data, or simply whichever one you feel more comfortable with.

Listing 9-14: **The 1998 season using external entity references for teams**

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE SEASON SYSTEM "baseball.dtd" [

  <!ENTITY angels SYSTEM "angels.xml">
  <!ENTITY astros SYSTEM "astros.xml">
  <!ENTITY athletics SYSTEM "athletics.xml">
  <!ENTITY bluejays SYSTEM "bluejays.xml">
  <!ENTITY braves SYSTEM "braves.xml">
  <!ENTITY brewers SYSTEM "brewers.xml">
  <!ENTITY cubs SYSTEM "cubs.xml">
```

Continued

Listing 9-14 *(continued)*

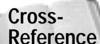
```
  <!ENTITY devilrays SYSTEM "devilrays.xml">
  <!ENTITY diamondbacks SYSTEM "diamondbacks.xml">
  <!ENTITY dodgers SYSTEM "dodgers.xml">
  <!ENTITY expos SYSTEM "expos.xml">
  <!ENTITY giants SYSTEM "giants.xml">
  <!ENTITY indians SYSTEM "indians.xml">
  <!ENTITY mariners SYSTEM "mariners.xml">
  <!ENTITY marlins SYSTEM "marlins.xml">
  <!ENTITY mets SYSTEM "mets.xml">
  <!ENTITY orioles SYSTEM "orioles.xml">
  <!ENTITY padres SYSTEM "padres.xml">
  <!ENTITY phillies SYSTEM "phillies.xml">
  <!ENTITY pirates SYSTEM "pirates.xml">
  <!ENTITY rangers SYSTEM "rangers.xml">
  <!ENTITY redsox SYSTEM "red sox.xml">
  <!ENTITY reds SYSTEM "reds.xml">
  <!ENTITY rockies SYSTEM "rockies.xml">
  <!ENTITY royals SYSTEM "royals.xml">
  <!ENTITY tigers SYSTEM "tigers.xml">
  <!ENTITY twins SYSTEM "twins.xml">
  <!ENTITY whitesox SYSTEM "whitesox.xml">
  <!ENTITY yankees SYSTEM "yankees.xml">

]>
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>National</LEAGUE_NAME>
    <DIVISION>
       <DIVISION_NAME>East</DIVISION_NAME>
        &marlins;
        &braves;
        &expos;
        &mets;
        &phillies;
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>Central</DIVISION_NAME>
        &cubs;
        &reds;
        &astros;
        &brewers;
        &pirates;
    </DIVISION>
    <DIVISION>
       <DIVISION_NAME>West</DIVISION_NAME>
       &diamondbacks;
       &rockies;
       &dodgers;
       &padres;
       &giants;
```

```
    </DIVISION>
  </LEAGUE>
  <LEAGUE>
    <LEAGUE_NAME>American</LEAGUE_NAME>
    <DIVISION>
      <DIVISION_NAME>East</DIVISION_NAME>
      &orioles;
      &redsox;
      &yankees;
      &devilrays;
      &bluejays
    </DIVISION>
    <DIVISION>
      <DIVISION_NAME>Central</DIVISION_NAME>
      &whitesox;
      &indians;
      &tigers;
      &royals;
      &twins;
    </DIVISION>
    <DIVISION>
      <DIVISION_NAME>West</DIVISION_NAME>
      &angels;
      &athletics;
      &mariners;
      &rangers;
     </DIVISION>
  </LEAGUE>
</SEASON>
```

A final, less likely, alternative is to actually build teams from external player entities into separate files and then combine those team files into the divisions, leagues, and seasons. The master document can define the entity references used in the child team documents. However, in this case the team documents are not usable on their own because the entity references are not defined until they're aggregated into the master document.

It's truly unfortunate that only the top-level document can be attached to a DTD. This somewhat limits the utility of external parsed entities. However, when you learn about XLinks and XPointers, you'll see some other ways to build large, compound documents out of small parts. However, those techniques are not part of the core XML standard and not necessarily supported by any validating XML processor and Web browser like the techniques of this chapter.



Chapter 16, *XLinks*, covers XLinks and Chapter 17, *XPointers*, discusses XPointers.

Entities and DTDs in Well-Formed Documents

Part I of this book explored well-formed XML documents without DTDs. And Part II has been exploring documents that have DTDs and adhere to the constraints in the DTD, that is valid documents. But there is a third level of conformance to the XML standard: documents that have DTDs and are well-formed but aren’t valid, either because the DTD is incomplete or because the document doesn’t fit the DTD’s constraints. This is the least common of the three types.

However, not all documents need to be valid. Sometimes it suffices for an XML document to be merely well-formed. DTDs also have a place in well-formed XML documents (though they aren’t required as they are for valid documents). And some non-validating XML processors can take advantage of information in a DTD without requiring perfect conformance to it. We explore that option in this section.

If a well-formed but invalid XML document does have a DTD, that DTD must have the same general form as explored in previous chapters. That is, it begins with a document type declaration and may contain `ELEMENT`, `ATTLIST`, and `ENTITY` declarations. Such a document differs from a valid document in that the processor only considers the `ENTITY` declarations.

Internal Entities

The primary advantage of using a DTD in invalid well-formed XML documents is that you may use internal general entity references other than the five pre-defined references `>`, `<`, `"`, `'`, and `&`. You simply declare the entities you want as normal; then use them in your document.

For example, to repeat the earlier example, suppose you want the entity reference `&ERH;` to be replaced by the string “Elliotte Rusty Harold” (OK, suppose *I* want the entity reference `&ERH;` to be replaced by the string “Elliotte Rusty Harold”) but you don’t want to write a complete DTD for your document. Simply declare the `ERH` entity reference in a DTD, as Listing 9-15 demonstrates. This document is only well-formed, not valid, but perfectly acceptable if you don’t require validity.

Listing 9-15: The ERH entity reference in a DTD yields a well-formed yet invalid document

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
   <!ENTITY ERH "Elliotte Rusty Harold">
]>
<DOCUMENT>
  <TITLE>&ERH;</TITLE>
```

```
    <SIGNATURE>
      <COPYRIGHT>1999 &ERH;</COPYRIGHT>
      <EMAIL>elharo@metalab.unc.edu</EMAIL>
      <LAST_MODIFIED>March 10, 1999</LAST_MODIFIED>
     </SIGNATURE>
  </DOCUMENT>
```

The document type declaration in Listing 9-15 is very sparse. Aside from defining the `ERH` entity reference, it simply says that the root element is `DOCUMENT`. However, well-formedness doesn't even require the document to adhere to that one small constraint. For example, Listing 9-16 displays another document that uses a `PAGE` root element even though the document type declaration still says the root element should be `DOCUMENT`. This document is still well-formed, but it's not valid-then again neither was Listing 9-15.

Listing 9-16: **A well-formed but invalid document**

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
   <!ENTITY ERH "Elliotte Rusty Harold">
]>
<PAGE>
  <TITLE>&ERH;</TITLE>
  <SIGNATURE>
    <COPYRIGHT>1999 &ERH;</COPYRIGHT>
    <EMAIL>elharo@metalab.unc.edu</EMAIL>
    <LAST_MODIFIED>March 10, 1999</LAST_MODIFIED>
   </SIGNATURE>
</PAGE>
```

It's possible that the DTD may contain other `<!ELEMENT>`, `<!ATTLIST>`, and `<!NOTATION>` declarations as well. All of these are ignored by a non-validating processor. Only `<!ENTITY>` declarations are considered. The DTD of Listing 9-17 actively contradicts its contents. For instance, the `ADDRESS` element is supposed to be empty according to the DTD but in fact contains several undeclared child elements. Furthermore, each `ADDRESS` element is required to have `OCCUPANT`, `STREET`, `CITY`, and `ZIP` attributes but these are nowhere to be found. The root element is supposed to be `DOCUMENT`, not `ADDRESS`. The `DOCUMENT` element should contain a `TITLE` and a `SIGNATURE`, neither of which is declared in the DTD. This document is still well-formed, though very, very invalid.

Listing 9-17: **An extremely invalid, though still well-formed, document**

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
   <!ENTITY ERH "Elliotte Rusty Harold">
   <!ELEMENT ADDRESS EMPTY>
   <!ELEMENT DOCUMENT (TITLE, ADDRESS+, SIGNATURE)>
   <!ATTLIST ADDRESS OCCUPANT   CDATA #REQUIRED>
   <!ATTLIST ADDRESS DEPARTMENT CDATA #IMPLIED>
   <!ATTLIST ADDRESS COMPANY    CDATA #IMPLIED>
   <!ATTLIST ADDRESS STREET     CDATA #REQUIRED>
   <!ATTLIST ADDRESS CITY       CDATA #REQUIRED>
   <!ATTLIST ADDRESS ZIP        CDATA #REQUIRED>
]>
<ADDRESS>
  <OCCUPANT>Elliotte Rusty Harold</OCCUPANT>
  <DEPARTMENT>Computer Science</DEPARTMENT>
  <COMPANY>Polytechnic University</COMPANY>
  <STREET>5 Metrotech Center</STREET>
  <CITY>Brooklyn</CITY>
  <STATE>NY</STATE>
  <ZIP>11201</ZIP>
</ADDRESS>
```

External Entities

Non-validating processors may resolve external entity references, but they are not required to. Expat, the open source XML parser used by Mozilla, for instance, does not resolve external entity references. Most others including the one used in Internet Explorer 5.0 do. Non-validating processors may only resolve parsed entities, however.They may not resolve unparsed external entities containing non-XML data such as images or sounds.

External entities are particularly useful for storing boilerplate text. For instance, HTML predefines entity references for the non-ASCII ISO Latin-1 letters that are a little easier to remember than the numeric character entity references. For instance, å is `˚`, _ is `þ`, _ is `Ý`, and so on. Listing 9-18 demonstrates an official ISO DTD that defines these references (with slight modifications to the comments and whitespace to make it fit neatly on the page).

Listing 9-18: A DTD for the non-ASCII ISO-Latin-1 characters

```
<!— (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
—>
<!— Character entity set. Typical invocation:
     <!ENTITY % ISOlat1 PUBLIC
       "ISO 8879-1986//ENTITIES Added Latin 1//EN//XML">
     %ISOlat1;
—>
<!— This version of the entity set can be used with any SGML
     document which uses ISO 8859-1 or ISO 10646 as its
     document character set. This includes XML documents and
     ISO HTML documents.

    Version: 1998-10-01
—>

<!ENTITY Agrave "&#192;" ><!— capital A, grave accent —>
<!ENTITY Aacute "&#193; "><!— capital A, acute accent —>
<!ENTITY Acirc  "&#194; "><!— capital A, circumflex accent —>
<!ENTITY Atilde "&#195; "><!— capital A, tilde —>
<!ENTITY Auml   "&#196; "><!— capital A, dieresis umlaut —>
<!ENTITY Aring  "&#197; "><!— capital A, ring —>
<!ENTITY AElig  "&#198; "><!— capital AE diphthong ligature—>
<!ENTITY Ccedil "&#199; "><!— capital C, cedilla —>
<!ENTITY Egrave "&#200; "><!— capital E, grave accent —>
<!ENTITY Eacute "&#201; "><!— capital E, acute accent —>
<!ENTITY Ecirc  "&#202; "><!— capital E, circumflex accent —>
<!ENTITY Euml   "&#203; "><!— capital E, dieresis umlaut —>
<!ENTITY Igrave "&#204; "><!— capital I, grave accent —>
<!ENTITY Iacute "&#205; "><!— capital I, acute accent —>
<!ENTITY Icirc  "&#206;" ><!— capital I, circumflex accent —>
<!ENTITY Iuml   "&#207;" ><!— capital I, dieresis umlaut —>
<!ENTITY ETH    "&#208;" ><!— capital Eth, Icelandic —>
<!ENTITY Ntilde "&#209;" ><!— capital N, tilde —>
<!ENTITY Ograve "&#210;" ><!— capital O, grave accent —>
<!ENTITY Oacute "&#211;" ><!— capital O, acute accent —>
<!ENTITY Ocirc  "&#212;" ><!— capital O, circumflex accent —>
<!ENTITY Otilde "&#213;" ><!— capital O, tilde —>
<!ENTITY Ouml   "&#214;"><!—capital O dieresis/umlaut mark—>
<!ENTITY Oslash "&#216;" ><!— capital O, slash —>
<!ENTITY Ugrave "&#217;" ><!— capital U, grave accent —>
<!ENTITY Uacute "&#218;" ><!— capital U, acute accent —>
<!ENTITY Ucirc  "&#219;" ><!— capital U circumflex accent —>
<!ENTITY Uuml   "&#220;" ><!— capital U dieresis umlaut —>
<!ENTITY Yacute "&#221;" ><!— capital Y, acute accent —>
<!ENTITY THORN  "&#222;" ><!— capital THORN, Icelandic —>
```

Continued

Listing 9-18 *(continued)*

```
<!ENTITY szlig  "&#223;" ><!– small sharp s, (sz ligature) –>
<!ENTITY agrave "&#224;" ><!– small a, grave accent –>
<!ENTITY aacute "&#225;" ><!– small a, acute accent –>
<!ENTITY acirc  "&#226;" ><!– small a, circumflex accent –>
<!ENTITY atilde "&#227;" ><!– small a, tilde –>
<!ENTITY auml   "&#228;" ><!– small a dieresis/umlaut mark–>
<!ENTITY aring  "&#229;" ><!– small a, ring –>
<!ENTITY aelig  "&#230;" ><!– small ae, diphthong ligature –>
<!ENTITY ccedil "&#231;" ><!– small c, cedilla –>
<!ENTITY egrave "&#232;" ><!– small e, grave accent –>
<!ENTITY eacute "&#233;" ><!– small e, acute accent –>
<!ENTITY ecirc  "&#234;" ><!– small e, circumflex accent –>
<!ENTITY euml   "&#235;" ><!– small e, dieresis or umlaut –>
<!ENTITY igrave "&#236;" ><!– small i, grave accent –>
<!ENTITY iacute "&#237;" ><!– small i, acute accent –>
<!ENTITY icirc  "&#238;" ><!– small i, circumflex accent –>
<!ENTITY iuml   "&#239;" ><!– small i, dieresis or umlaut –>
<!ENTITY eth    "&#240;" ><!– small eth, Icelandic –>
<!ENTITY ntilde "&#241;" ><!– small n, tilde –>
<!ENTITY ograve "&#242;" ><!– small o, grave accent –>
<!ENTITY oacute "&#243;" ><!– small o, acute accent –>
<!ENTITY ocirc  "&#244;" ><!– small o, circumflex accent –>
<!ENTITY otilde "&#245;" ><!– small o, tilde –>
<!ENTITY ouml   "&#246;" ><!– small o, dieresis or umlaut–>
<!ENTITY oslash "&#248;" ><!– small o, slash –>
<!ENTITY ugrave "&#249;" ><!– small u, grave accent –>
<!ENTITY uacute "&#250;" ><!– small u, acute accent –>
<!ENTITY ucirc  "&#251;" ><!– small u, circumflex accent –>
<!ENTITY uuml   "&#252;" ><!– small u, dieresis or umlaut –>
<!ENTITY yacute "&#253;" ><!– small y, acute accent –>
<!ENTITY thorn  "&#254;" ><!– small thorn, Icelandic –>
<!ENTITY yuml   "&#255;" ><!– small y, dieresis or umlaut –>
```

Rather than including Listing 9-18 in the internal subset of your document's DTD, you can simply use a parameter entity reference to link to it, then use the general entity references in your document.

For example, suppose you wanted to put the medieval document Hildebrandslied on the Web in well-formed XML. However since this manuscript is written in German, it uses the non-ASCII characters ê, î, ô, û, and æ.

For maximum portability you can type the poem in ASCII while encoding these letters as the entity references ê, î, ô û, and æ respectively. However, even if you don't require a valid finished document, you still need a DTD to

declare these and any other entity references you may use. The simplest way to get the extra characters you need, is merely to refer to the external DTD of Listing 9-18. Listing 9-19 demonstrates

:Listing 9-19: A well-formed, invalid document that uses entity references for non ASCII ISO-Latin-1 characters

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [
  <!ENTITY % ISOlat1
     PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN//XML"
            "http://www.schema.net/public-text/ISOlat1.pen">
     %ISOlat1;
]>
<DOCUMENT>
  <TITLE>Das Hildebrandslied, circa 775 C.E. </TITLE>
  <LINE>Ik gih&ocirc;rta dhat seggen,</LINE>
  <LINE>dhat sih urh&ecirc;ttun &aelig;non muot&icirc;n,</LINE>
  <LINE>Hiltibrant enti Hadhubrant untar heriun tu&ecirc;m.
  </LINE>
  <LINE>sunufatarungo: iro saro rihtun,</LINE>
  <COMMENT>I'll spare you the next 61 lines</COMMENT>
</DOCUMENT>
```

The document part consists of well-formed XML using tags made up on the spot. These are not declared in the DTD and do not need to be for a merely well-formed document. However the entity references do need to be declared in the DTD, either in the internal or external subset. Listing 9-19 declares them in the external subset by using the external parameter entity reference `%ISOlat1` to load the entities declared in Listing 9-18.

DTDs are also useful for storing common boilerplate text used across a Web site of well- formed XML documents, much as they are for valid XML documents. The procedure is a little easier when working with merely well formed XML documents, because there’s no chance that the boilerplate you insert will not meet the constraints of the parent document DTD.

First, place the boilerplate in a file without a DTD, as shown in Listing 9-20.

Listing 9-20: **Signature boilerplate without a DTD**

```
<?xml version="1.0"?>
<SIGNATURE>
  <COPYRIGHT>1999 Elliotte Rusty Harold</COPYRIGHT>
  <EMAIL>elharo@metalab.unc.edu</EMAIL>
</SIGNATURE>
```

Next, write a small DTD as in Listing 9-21 that defines an entity reference for the file in Listing 9-20. Here, I assume that you can locate Listing 9-20 in the file signature.xml in the boilerplate directory at the root level of the Web server, and that you can find Listing 9-21 in the file signature.dtd in the dtds directory at the root level of the Web server.

Listing 9-21: **Signature DTD that defines an entity reference**

```
<!ENTITY SIGNATURE SYSTEM "/boilerplate/signature.xml">
```

Now, you can import signature.dtd in any document, then use the general entity reference `&SIGNATURE;` to embed the contents of signature.xml in your file. Listing 9-22 demonstrates.

Listing 9-22: **A file that uses &SIGNATURE;**

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
   <!ENTITY % SIG SYSTEM "/dtds/signature.dtd">
   %SIG;
]>
<DOCUMENT>
  <TITLE>A Very Boring Document</TITLE>
  &SIGNATURE;
</DOCUMENT>
```

This may seem like one more level of indirection than you really need. For instance, Listing 9-23 defines the `&SIGNATURE;` entity reference directly in its internal DTD subset, and indeed this does work. However, the additional level of indirection

provides protection against a reorganization of a Web site since you cannot only change the signature used on all your pages by editing one file. You can also change the location of the signature used by all your Web pages by editing one file. On the other hand, the more direct approach of Listing 9-22 more easily allows for different signatures on different pages.

Listing 9-23: A file that uses &SIGNATURE; with one less level of indirection

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
  <!ENTITY SIGNATURE SYSTEM "/boilerplate/signature.xml">
]>
<DOCUMENT>
  <TITLE>A Very Boring Document</TITLE>
  &SIGNATURE;
</DOCUMENT>
```

Summary

In this chapter, you discovered that XML documents are built from both internal and external entities. . In particular, you learned the following:

- ✦ Entities are the physical storage units from which the document is assembled.
- ✦ An entity holds content: well-formed XML, other forms of text, or binary data.
- ✦ Internal entities are defined completely within the document and external entities draw their content from another resource located via a URL.
- ✦ General entity references have the form `&name;` and are used in a document's content.
- ✦ Internal general entity references are replaced by an entity value given in the entity declaration.
- ✦ External general entity references are replaced by the data at a URL specified in the entity declaration after the `SYSTEM` keyword.
- ✦ Internal parameter entity references have the form `%name;` and are used exclusively in DTDs.
- ✦ You can merge different DTDs with external parameter entity references.
- ✦ External entity references enable you to build large, compound documents out of small parts.

✦ There is a third level of conformance to the XML standard: well-formed, but not valid. This is either because the DTD is incomplete or because the document doesn't meet the DTD's constraints.

When a document uses attributes, the attributes must also be declared in the DTD. The next chapter discusses how to declare attributes in DTDs, and how you can thereby attach constraints to the attribute values.