

# Document Type Definitions and Validity

---

**X**ML has been described as a meta-markup language, that is, a language for describing markup languages. In this chapter you begin to learn how to document and describe the new markup languages you create. Such markup languages (also known as *tag sets*) are defined via a document type definition (DTD), which is what this chapter is all about. Individual documents can be compared against DTDs in a process known as validation. If the document matches the constraints listed in the DTD, then the document is said to be valid. If it doesn't, the document is said to be invalid.

## Document Type Definitions

The acronym DTD stands for *document type definition*. A document type definition provides a list of the elements, attributes, notations, and entities contained in a document, as well as their relationships to one another. DTDs specify a set of rules for the structure of a document. For example, a DTD may dictate that a `BOOK` element have exactly one `ISBN` child, exactly one `TITLE` child, and one or more `AUTHOR` children, and it may or may not contain a single `SUBTITLE`. The DTD accomplishes this with a list of markup declarations for particular elements, entities, attributes, and notations.

Cross-Reference

This chapter focuses on element declarations. Chapters 9, 10, and 11 introduce entities, attributes, and notations, respectively.

DTDs can be included in the file that contains the document they describe, or they can be linked from an external URL.



### In This Chapter

Document Type Definitions (DTDs)

Document type declarations

Validation against a DTD

The list of elements

Element declarations

Comments in DTDs

Common DTDs that can be shared among documents



Such external DTDs can be shared by different documents and Web sites. DTDs provide a means for applications, organizations, and interest groups to agree upon, document, and enforce adherence to markup standards.

For example, a publisher may want an author to adhere to a particular format because it makes it easier to lay out a book. An author may prefer writing words in a row without worrying about matching up each bullet point in the front of the chapter with a subhead inside the chapter. If the author writes in XML, it's easy for the publisher to check whether the author adhered to the predetermined format specified by the DTD, and even to find out exactly where and how the author deviated from the format. This is much easier than having editors read through documents with the hope that they spot all the minor deviations from the format, based on style alone.

DTDs also help ensure that different people and programs can read each other's files. For instance, if chemists agree on a single DTD for basic chemical notation, possibly via the intermediary of an appropriate professional organization such as the American Chemical Society, then they can be assured that they can all read and understand one another's papers. The DTD defines exactly what is and is not allowed to appear inside a document. The DTD establishes a standard for the elements that viewing and editing software must support. Even more importantly, it establishes extensions beyond those that the DTD declares are invalid. Thus, it helps prevent software vendors from embracing and extending open protocols in order to lock users into their proprietary software.

Furthermore, a DTD shows how the different elements of a page are arranged without actually providing their data. A DTD enables you to see the structure of your document separate from the actual data. This means you can slap a lot of fancy styles and formatting onto the underlying structure without destroying it, much as you paint a house without changing its basic architectural plan. The reader of your page may not see or even be aware of the underlying structure, but as long as it's there, human authors and JavaScripts, CGIs, servlets, databases, and other programs can use it.

There's more you can do with DTDs. You can use them to define glossary entities that insert boilerplate text such as a signature block or an address. You can ascertain that data entry clerks are adhering to the format you need. You can migrate data to and from relational and object databases. You can even use XML as an intermediate format to convert different formats with suitable DTDs. So let's get started and see what DTDs really look like.

## Document Type Declarations

A *document type declaration* specifies the DTD a document uses. The document type declaration appears in a document's prolog, after the XML declaration but before the root element. It may contain the document type definition or a URL identifying the file where the document type definition is found. It may even contain both, in

which case the document type definition has two parts, the internal and external subsets.



Caution

A document type *declaration* is not the same thing as a document type *definition*. Only the document type definition is abbreviated DTD. A document type declaration must contain or refer to a document type definition, but a document type definition never contains a document type declaration. I agree that this is unnecessarily confusing. Unfortunately, XML seems stuck with this terminology. Fortunately, most of the time the difference between the two is not significant.

Recall Listing 3-2 (greeting.xml) from Chapter 3. It is shown below:

```
<?xml version="1.0" standalone="yes"?>
<GREETING>
Hello XML!
</GREETING>
```

This document contains a single element, GREETING. (Remember, `<?xml version="1.0" standalone="yes"?>` is a processing instruction, not an element.) Listing 8-1 shows this document, but now with a document type declaration. The document type declaration declares that the root element is GREETING. The document type declaration also contains a document type definition, which declares that the GREETING element contains parsed character data.

### Listing 8-1: Hello XML with DTD

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE GREETING [
  <!ELEMENT GREETING (#PCDATA)>
]>
<GREETING>
Hello XML!
</GREETING>
```

The only difference between Listing 3-2 and Listing 8-1 are the three new lines added to Listing 8-1:

```
<!DOCTYPE GREETING [
  <!ELEMENT GREETING (#PCDATA)>
]>
```

These lines are this Listing 8-1's document type declaration. The document type declaration comes between the XML declaration and the document itself. The XML declaration and the document type declaration together are called the *prolog* of the document. In this short example, `<?xml version="1.0" standalone="yes"?>` is the XML declaration; `<!DOCTYPE GREETING [ <!ELEMENT GREETING (#PCDATA)> ]>` is the document type declaration; `<!ELEMENT GREETING (#PCDATA)>` is the

document type definition; and `<GREETING> Hello XML! </GREETING>` is the document or root element.

A document type declaration begins with `<!DOCTYPE` and ends with `>`. It's customary to place the beginning and end on separate lines, but line breaks and extra whitespace are not significant. The same document type declaration could be written on a single line:

```
<!DOCTYPE GREETING [ <!ELEMENT GREETING (#PCDATA)> ]>
```

The name of the root element—`GREETING` in this example follows `<!DOCTYPE`. This is not just a name but a requirement. Any valid document with this document type declaration must have the root element `GREETING`. In between the `[` and the `]` is the document type definition.

The DTD consists of a series of markup declarations that declare particular elements, entities, and attributes. One of these declarations declares the root element. In Listing 8-1 the entire DTD is simply this one line:

```
<!ELEMENT GREETING (#PCDATA)>
```

In general, of course, DTDs will be much longer and more complex.

The single line `<!ELEMENT GREETING (#PCDATA)>` (case-sensitive as most things are in XML) is an *element type declaration*. In this case, the name of the declared element is `GREETING`. It is the only element. This element may contain parsed character data (or `#PCDATA`). Parsed character data is essentially any text that's not markup text. This also includes entity references, such as `&amp;`, that are replaced by text when the document is parsed.

You can load this document into an XML browser as usual. Figure 8-1 shows Listing 8-1 in Internet Explorer 5.0. The result is probably what you'd expect, a collapsible outline view of the document source. Internet Explorer indicates that a document type declaration is present by adding the line `<!DOCTYPE GREETING (View Source for full doctype...)>` in blue.

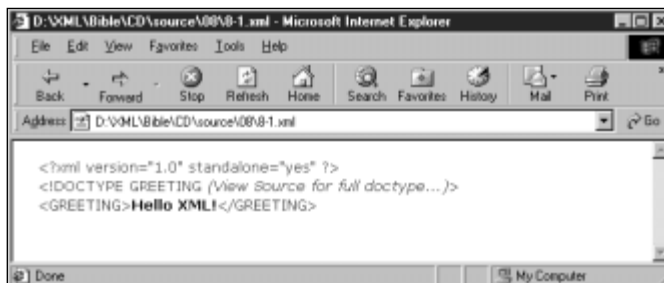


Figure 8-1: Hello XML with DTD displayed in Internet Explorer 5.0

Of course, the document can be combined with a style sheet just as it was in Listing 3-6 in Chapter 3. In fact, you can use the same style sheet. Just add the usual `<?xml-stylesheet?>` processing instruction to the prolog as shown in Listing 8-2.

### Listing 8-2: Hello XML with a DTD and style sheet

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="greeting.css"?>
<!DOCTYPE GREETING [
  <!ELEMENT GREETING (#PCDATA)>
]>
<GREETING>
Hello XML!
</GREETING>
```

Figure 8-2 shows the resulting Web page. This is *exactly* the same as it was in Figure 3-3 in Chapter 3 without the DTD. Formatting generally does not consider the DTD.



Figure 8-2 Hello XML with a DTD and style sheet displayed in Internet Explorer 5.0

## Validating Against a DTD

A valid document must meet the constraints specified by the DTD. Furthermore, its root element must be the one specified in the document type declaration. What the document type declaration and DTD in Listing 8-1 say is that a valid document must look like this:

```
<GREETING>
  various random text but no markup
</GREETING>
```

A valid document may not look like this:

```
<GREETING>
  <sometag>various random text</sometag>
  <someEmptyTag/>
</GREETING>
```

Nor may it look like this:

```
<GREETING>
  <GREETING>various random text</GREETING>
</GREETING>
```

This document must consist of nothing more and nothing less than parsed character data between an opening `<GREETING>` tag and a closing `</GREETING>` tag. Unlike a merely well-formed document, a valid document does not allow arbitrary tags. Any tags used must be declared in the document's DTD. Furthermore, they must be used only in the way permitted by the DTD. In Listing 8-1, the `<GREETING>` tag can be used only to start the root element, and it may not be nested.

Suppose we make a simple change to Listing 8-2 by replacing the `<GREETING>` and `</GREETING>` tags with `<foo>` and `</foo>`, as shown in Listing 8-3. Listing 8-3 is *invalid*. It is a well-formed XML document, but it does not meet the constraints specified by the document type declaration and the DTD it contains.

### Listing 8-3: Invalid Hello XML does not meet DTD rules

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="greeting.css"?>
<!DOCTYPE GREETING [
  <!ELEMENT GREETING (#PCDATA)>
]>
<foo>
Hello XML!
</foo>
```

**Note**

Not all documents have to be valid, and not all parsers check documents for validity. In fact, most Web browsers including IE5 and Mozilla do not check documents for validity.

A validating parser reads a DTD and checks whether a document adheres to the rules specified by the DTD. If it does, the parser passes the data along to the XML application (such as a Web browser or a database). If the parser finds a mistake, then it reports the error. If you're writing XML by hand, you'll want to validate your

documents before posting them so you can be confident that readers won't encounter errors.

There are about a dozen different validating parsers available on the Web. Most of them are free. Most are libraries intended for programmers to incorporate into their own, more finished products, and they have minimal (if any) user interfaces. Parsers in this class include IBM's alphaWorks' XML for Java, Microsoft and DataChannel's XJParser, and Silfide's SXP.

**XML for Java:** <http://www.alphaworks.ibm.com/tech/xml>

**XJParser:** [http://www.datachannel.com/xml\\_resources/](http://www.datachannel.com/xml_resources/)

**SXP:** <http://www.loria.fr/projets/XSilfide/EN/sxp/>

Some libraries also include stand-alone parsers that run from the command line. These are programs that read an XML file and report any errors found but do not display them. For example, XJParse is a Java program included with IBM's XML for Java 1.1.16 class library in the `samples.XJParse` package. To run this program, you first have to add the XML for Java jar files to your Java class path. You can then validate a file by opening a DOS Window or a shell prompt and passing the local name or remote URL of the file you want to validate to the XJParse program, like this:

```
C:\xml4j>java samples.XJParse.XJParse -d D:\XML\08\invalid.xml
```



Note

At the time of this writing IBM's alphaWorks released version 2.0.6 of XML for Java. In this version you invoke only XJParse instead of `samples.XJParse`. However, version 1.1.16 provides more features for stand-alone validation.

You can use a URL instead of a file name, as shown below:

```
C:\xml4j>java samples.XJParse.XJParse -d
http://metalab.unc.edu/books/bible/examples/08/invalid.xml
```

In either case, XJParse responds with a list of the errors found, followed by a tree form of the document. For example:

```
D:\XML\07\invalid.xml: 6, 4: Document root element, "foo", must
match DOCTYPE root, "GREETING".
D:\XML\07\invalid.xml: 8, 6: Element "<foo>" is not valid in
this context.
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="greeting.css"?>
<!DOCTYPE GREETING [
  <!ELEMENT GREETING (#PCDATA)>
]>
<foo>
Hello XML!
</foo>
```

This is not especially attractive output. However, the purpose of a validating parser such as XJParse isn't to display XML files. Instead, the parser's job is to divide the document into a tree structure and pass the nodes of the tree to the program that will display the data. This might be a Web browser such as Netscape Navigator or Internet Explorer. It might be a database. It might even be a custom program you've written yourself. You use XJParse, or other command line, validating parser to verify that you've written good XML that other programs can handle. In essence, this is a proofreading or quality assurance phase, not finished output.

Because XML for Java and most other validating parsers are written in Java, they share all the disadvantages of cross-platform Java programs. First, before you can run the parser you must have the Java Development Kit (JDK) or Java Runtime Environment installed. Secondly, you need to add the XML for Java jar files to your class path. Neither of these tasks is as simple as it should be. None of these tools were designed with an eye toward nonprogrammer end-users; they tend to be poorly designed and frustrating to use.

If you're writing documents for Web browsers, the simplest way to validate them is to load them into the browser and see what errors it reports. However, not all Web browsers validate documents. Some may merely accept well-formed documents without regard to validity. Internet Explorer 5.0 beta 2 validated documents, but the release version did not.

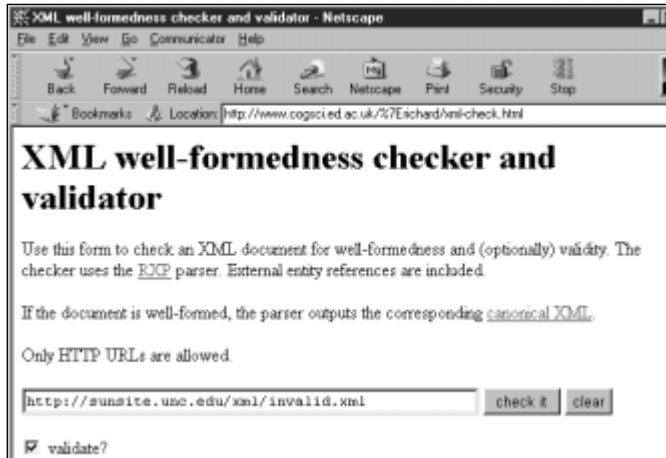


The JRE for Windows and Unix is included on the CD-ROM in the misc/jre folder.

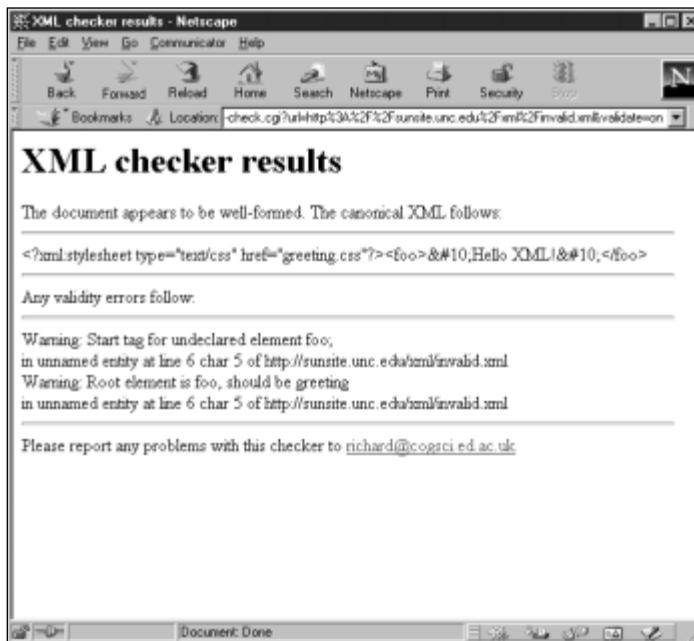
Web-based validators are an alternative if the documents are placed on a Web server and aren't particularly private. These parsers only require that you enter the URL of your document in a simple form. They have the distinct advantage of not requiring you to muck around with Java runtime software, class paths, and environment variables.

Richard Tobin's RXP-based, Web-hosted XML well-formedness checker and validator is shown in Figure 8-3. You'll find it at <http://www.cogsci.ed.ac.uk/%7Erichard/xml-check.html>. Figure 8-4 shows the errors displayed as a result of using this program to validate Listing 8-3.

Brown University's Scholarly Technology Group provides a validator at <http://www.stg.brown.edu/service/xmlvalid/> that's notable for allowing you to upload files from your computer instead of placing them on a public Web server. This is shown in Figure 8-5. Figure 8-6 shows the results of using this program to validate Listing 8-3.



**Figure 8-3:** Richard Tobin's RXP-based, Web-hosted XML well-formedness checker and validator



**Figure 8-4:** The errors with Listing 8-3, as reported by Richard Tobin's XML validator

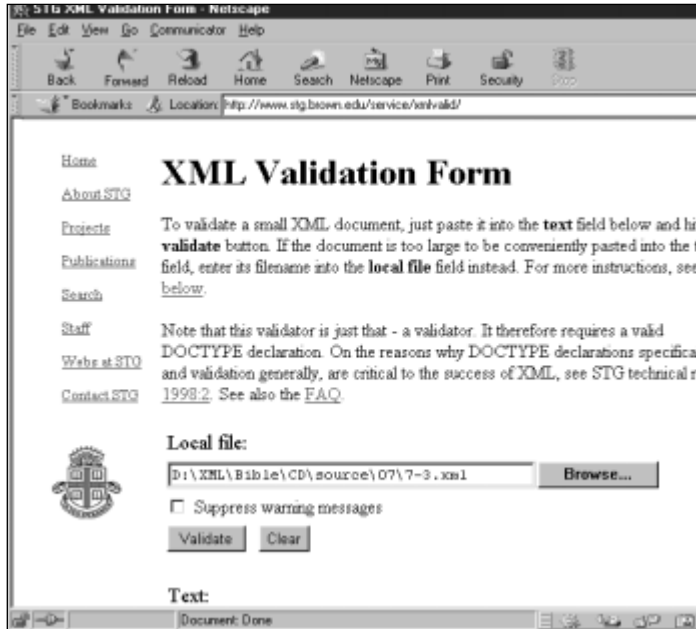


Figure 8-5: Brown University's Scholarly Technology Group's Web-hosted XML validator

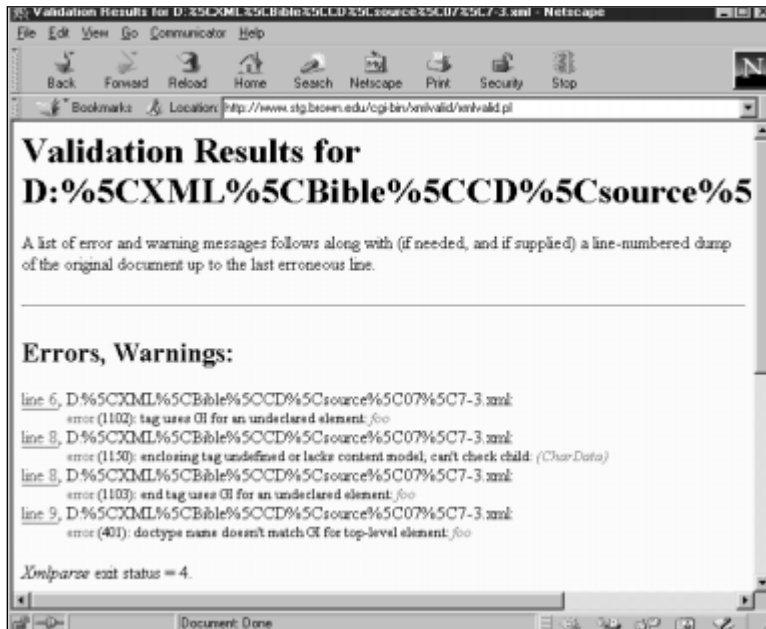


Figure 8-6: The errors with Listing 8-3, as reported by Brown University's Scholarly Technology Group's XML validator

## Listing the Elements

The first step to creating a DTD appropriate for a particular document is to understand the structure of the information you'll encode using the elements defined in the DTD. Sometimes information is quite structured, as in a contact list. Other times it is relatively free-form, as in an illustrated short story or a magazine article.

Let's use a relatively structured document as an example. In particular, let's return to the baseball statistics first shown in Chapter 4. Adding a DTD to that document enables us to enforce constraints that were previously adhered to only by convention. For instance, we can require that a SEASON contain exactly two LEAGUE children, every TEAM have a TEAM\_CITY and a TEAM\_NAME, and the TEAM\_CITY always precede the TEAM\_NAME.

Recall that a complete baseball statistics document contains the following elements:

|               |                   |
|---------------|-------------------|
| SEASON        | RBI               |
| YEAR          | STEALS            |
| LEAGUE        | CAUGHT_STEALING   |
| LEAGUE_NAME   | SACRIFICE_HITS    |
| DIVISION      | SACRIFICE_FLIES   |
| DIVISION_NAME | ERRORS            |
| TEAM          | WALKS             |
| TEAM_CITY     | STRUCK_OUT        |
| TEAM_NAME     | HIT_BY_PITCH      |
| PLAYER        | COMPLETE_GAMES    |
| SURNAME       | SHUT_OUTS         |
| GIVEN_NAME    | ERA               |
| POSITION      | INNINGS           |
| GAMES         | HOME_RUNS         |
| GAMES_STARTED | RUNS              |
| AT_BATS       | EARNED_RUNS       |
| RUNS          | HIT_BATTER        |
| HITS          | WILD_PITCHES      |
| DOUBLES       | BALK              |
| TRIPLES       | WALKED_BATTER     |
| HOME_RUNS     | STRUCK_OUT_BATTER |

|        |                |
|--------|----------------|
| WINS   | COMPLETE_GAMES |
| LOSSES | SHUT_OUTS      |
| SAVES  |                |

The DTD you write needs element declarations for each of these. Each element declaration lists the name of an element and the children the element may have. For instance, a DTD can require that a LEAGUE have exactly three DIVISION children. It can also require that the SURNAME element be inside a PLAYER element, never outside. It can insist that a DIVISION have an indefinite number of TEAM elements but never less than one.

A DTD can require that a PLAYER have exactly one each of the GIVEN\_NAME, SURNAME, POSITION, and GAMES elements, but make it optional whether a PLAYER has an RBI or an ERA. Furthermore, it can require that the GIVEN\_NAME, SURNAME, POSITION, and GAMES elements be used in a particular order. A DTD can also require that elements occur in a particular context. For instance, the GIVEN\_NAME, SURNAME, POSITION, and GAMES may be used only inside a PLAYER element.

It's often easier to begin if you have a concrete, well-formed example document in mind that uses all the elements you want in your DTD. The examples in Chapter 4 serve that purpose here. Listing 8-4 is a trimmed-down version of Listing 4-1 in Chapter 4. Although it has only two players, it demonstrates all the essential elements.

#### Listing 8-4: A well-formed XML document for which a DTD will be written

```
<?xml version="1.0" standalone="yes"?>
<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>
        <PLAYER>
          <SURNAME>Ludwick</SURNAME>
          <GIVEN_NAME>Eric</GIVEN_NAME>
          <POSITION>Starting Pitcher</POSITION>
          <WINS>1</WINS>
          <LOSSES>4</LOSSES>
          <SAVES>0</SAVES>
          <GAMES>13</GAMES>
        </PLAYER>
      </TEAM>
    </DIVISION>
  </LEAGUE>
</SEASON>
```

```

<GAMES_STARTED>6</GAMES_STARTED>
<COMPLETE_GAMES>0</COMPLETE_GAMES>
<SHUT_OUTS>0</SHUT_OUTS>
<ERA>7.44</ERA>
<INNINGS>32.2</INNINGS>
<HOME_RUNS>46</HOME_RUNS>
<RUNS>7</RUNS>
<EARNED_RUNS>31</EARNED_RUNS>
<HIT_BATTER>27</HIT_BATTER>
<WILD_PITCHES>0</WILD_PITCHES>
<BALK>2</BALK>
<WALKED_BATTER>0</WALKED_BATTER>
<STRUCK_OUT_BATTER>17</STRUCK_OUT_BATTER>
</PLAYER>
<PLAYER>
  <SURNAME>Daubach</SURNAME>
  <GIVEN_NAME>Brian</GIVEN_NAME>
  <POSITION>First Base</POSITION>
  <GAMES>10</GAMES>
  <GAMES_STARTED>3</GAMES_STARTED>
  <AT_BATS>15</AT_BATS>
  <RUNS>0</RUNS>
  <HITS>3</HITS>
  <DOUBLES>1</DOUBLES>
  <TRIPLES>0</TRIPLES>
  <HOME_RUNS>0</HOME_RUNS>
  <RBI>3</RBI>
  <STEALS>0</STEALS>
  <CAUGHT_STEALING>0</CAUGHT_STEALING>
  <SACRIFICE_HITS>0</SACRIFICE_HITS>
  <SACRIFICE_FLIES>0</SACRIFICE_FLIES>
  <ERRORS>0</ERRORS>
  <WALKS>1</WALKS>
  <STRUCK_OUT>5</STRUCK_OUT>
  <HIT_BY_PITCH>1</HIT_BY_PITCH>
</PLAYER>
</TEAM>
<TEAM>
  <TEAM_CITY>Montreal</TEAM_CITY>
  <TEAM_NAME>Expos</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>New York</TEAM_CITY>
  <TEAM_NAME>Mets</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>Philadelphia</TEAM_CITY>
  <TEAM_NAME>Phillies</TEAM_NAME>
</TEAM>

```

*Continued*

## Listing 8-4 (continued)

```
</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>
    </TEAM>
  </DIVISION>
  <DIVISION>
    <DIVISION_NAME>West</DIVISION_NAME>
    <TEAM>
      <TEAM_CITY>Anaheim</TEAM_CITY>
      <TEAM_NAME>Angels</TEAM_NAME>
    </TEAM>
  </DIVISION>
</LEAGUE>
</SEASON>
```

---

Table 8-1 lists the different elements in this particular listing, as well as the conditions they must adhere to. Each element has a list of the other elements it must contain, the other elements it may contain, and the element in which it must be contained. In some cases, an element may contain more than one child element of the same type. A `SEASON` contains one `YEAR` and two `LEAGUE` elements. A `DIVISION` generally contains more than one `TEAM`. Less obviously, some batters alternate between designated hitter and the outfield from game to game. Thus, a single `PLAYER` element might have more than one `POSITION`. In the table, a requirement for a particular number of children is indicated by prefixing the element with a number (for example, `2 LEAGUE`) and the possibility of multiple children is indicated by adding to the end of the element's name, such as `PLAYER(s)`.

Listing 8-4 adheres to these conditions. It could be shorter if the two `PLAYER` elements and some `TEAM` elements were omitted. It could be longer if many other `PLAYER` elements were included. However, all the other elements are required to be in the positions in which they appear.

Note

Elements have two basic types in XML. Simple elements contain text, also known as parsed character data, `#PCDATA` or `PCDATA` in this context. Compound elements contain other elements or, more rarely, text and other elements. There are no integer, floating point, date, or other data types in standard XML. Thus, you can't use a DTD to say that the number of walks must be a non-negative integer, or that the ERA must be a floating point number between 0.0 and 1.0, even though doing so would be useful in examples like this one. There are some early efforts to define schemas that use XML syntax to describe information that might traditionally be encoded in a DTD, as well as data type information. As of mid-1999, these are mostly theoretical with few practical implementations.

Now that you've identified the information you're storing, and the optional and required relationships between these elements, you're ready to build a DTD for the document that concisely — if a bit opaquely — summarizes those relationships.

It's often possible and convenient to cut and paste from one DTD to another. Many elements can be reused in other contexts. For instance, the description of a `TEAM` works equally well for football, hockey, and most other team sports.

You can include one DTD within another so that a document draws tags from both. You might, for example, use a DTD that describes the statistics of individual players in great detail, and then nest that DTD inside the broader DTD for team sports. To change from baseball to football, simply swap out your baseball player DTD for a football player DTD.

Cross-Reference

To do this, the file containing the DTD is defined as an external entity. External parameter entity references are discussed in Chapter 9, *Entities*.

**Table 8-1**  
**The Elements in the Baseball Statistics**

| <i>Element</i> | <i>Elements It Must Contain</i>             | <i>Elements It May Contain</i>   | <i>Element (if any) in Which It Must Be Contained</i> |
|----------------|---|--|---|
| SEASON         | YEAR,                                       | 2 LEAGUE   |   |
| YEAR           | Text  |  | SEASON  |
| LEAGUE         | LEAGUE_NAME,<br>3 DIVISION                  |  | SEASON  |
| LEAGUE_NAME    | Text  |  | LEAGUE  |
| DIVISION       | DIVISION_NAME<br>, TEAM                     | TEAM(s)  | LEAGUE  |
| DIVISION_NAME  | Text  |  | DIVISION  |
| TEAM           | TEAM_CITY,<br>TEAM_NAME                     | PLAYER(s)  | DIVISION  |
| TEAM_CITY      | Text  |  | TEAM  |
| TEAM_NAME      | Text  |  | TEAM  |
| PLAYER         | SURNAME, GIVEN<br>_NAME, POSITION,<br>GAMES | GAMES_STARTED, AT<br>_BATS, RUNS, HITS,<br>DOUBLES, TRIPLES,<br>HOME_RUNS, RBI,<br>STEALS, CAUGHT_<br>STEALING,<br>SACRIFICE_HITS,<br>SACRIFICE_FLIES,<br>ERRORS, WALKS,<br>STRUCK_OUT, HIT_<br>BY_PITCH, COMPLETE<br>_GAMES, SHUT_OUTS,<br>ERA, INNINGS, HIT_<br>BATTER, WILD_<br>PITCHES, BALK,<br>WALKED_BATTER,<br>STRUCK_OUT_<br>BATTER | TEAM  |
| SURNAME        | Text  |  | PLAYER  |
| GIVEN_NAME     | Text  |  | PLAYER  |
| POSITION       | Text  |  | PLAYER  |

| <i>Element</i>    | <i>Elements It Must Contain</i> | <i>Elements It May Contain</i> | <i>Element (if any) in Which It Must Be Contained</i> |
|-------------------|---------------------------------|--------------------------------|---|
| GAMES             | Text                            |                                | PLAYER  |
| GAMES_STARTED     | Text                            |                                | PLAYER  |
| AT_BATS           | Text                            |                                | PLAYER  |
| RUNS              | Text                            |                                | PLAYER  |
| HITS              | Text                            |                                | PLAYER  |
| DOUBLES           | Text                            |                                | PLAYER  |
| TRIPLES           | Text                            |                                | PLAYER  |
| HOME_RUNS         | Text                            |                                | PLAYER  |
| RBI               | Text                            |                                | PLAYER  |
| STEALS            | Text                            |                                | PLAYER  |
| CAUGHT_STEALING   | Text                            |                                | PLAYER  |
| SACRIFICE_HITS    | Text                            |                                | PLAYER  |
| SACRIFICE_FLIES   | Text                            |                                | PLAYER  |
| ERRORS            | Text                            |                                | PLAYER  |
| WALKS             | Text                            |                                | PLAYER  |
| STRUCK_OUT        | Text                            |                                | PLAYER  |
| HIT_BY_PITCH      | Text                            |                                | PLAYER  |
| COMPLETE_GAMES    | Text                            |                                | PLAYER  |
| SHUT_OUTS         | Text                            |                                | PLAYER  |
| ERA               | Text                            |                                | PLAYER  |
| INNINGS           | Text                            |                                | PLAYER  |
| HOME_RUNS_AGAINST | Text                            |                                | PLAYER  |

*Continued*

Table 8-1 (continued)

| <i>Element</i>    | <i>Elements It Must Contain</i> | <i>Elements It May Contain</i> | <i>Element (if any) in Which It Must Be Contained</i> |
|-------------------|---------------------------------|--------------------------------|---|
| RUNS_AGAINST      | Text                            |                                | PLAYER  |
| HIT_BATTER        | Text                            |                                | PLAYER  |
| WILD_PITCHES      | Text                            |                                | PLAYER  |
| BALK              | Text                            |                                | PLAYER  |
| WALKED_BATTER     | Text                            |                                | PLAYER  |
| STRUCK_OUT_BATTER | Text                            |                                | PLAYER  |

## Element Declarations

Each tag used in a valid XML document must be declared with an element declaration in the DTD. An element declaration specifies the name and possible contents of an element. The list of contents is sometimes called the content specification. The content specification uses a simple grammar to precisely specify what is and isn't allowed in a document. This sounds complicated, but all it really means is that you add a punctuation mark such as \*, ?, or + to an element name to indicate that it may occur more than once, may or may not occur, or must occur at least once.

DTDs are conservative. Everything not explicitly permitted is forbidden. However, DTD syntax does enable you to compactly specify relationships that are cumbersome to specify in sentences. For instance, DTDs make it easy to say that GIVEN\_NAME **must come before** SURNAME — which **must come before** POSITION, which **must come before** GAMES, which **must come before** GAMES\_STARTED, which **must come before** AT\_BATS, which **must come before** RUNS, which **must come before** HITS — and that all of these may appear only inside a PLAYER.

It's easiest to build DTDs hierarchically, working from the outside in. This enables you to build a sample document at the same time you build the DTD to verify that the DTD is itself correct and actually describes the format you want.

## ANY

The first thing you have to do is identify the root element. In the baseball example, SEASON is the root element. The !DOCTYPE declaration specifies this:

```
<!DOCTYPE SEASON [  
  
]>
```

However, this merely says that the root tag is SEASON. It does not say anything about what a SEASON element may or may not contain, which is why you must next declare the SEASON element in an element declaration. That's done with this line of code:

```
<!ELEMENT SEASON ANY>
```

All element type declarations begin with <!ELEMENT (case sensitive) and end with >. They include the name of the element being declared (SEASON in this example) followed by the content specification. The ANY keyword (again case-sensitive) says that all possible elements as well as parsed character data can be children of the SEASON element.

Using ANY is common for root elements — especially of unstructured documents — but should be avoided in most other cases. Generally it's better to be as precise as possible about the content of each tag. DTDs are usually refined throughout their development, and tend to become less strict over time as they reflect uses and contexts unimagined in the first cut. Therefore, it's best to start out strict and loosen things up later.

## #PCDATA

Although any element may appear inside the document, elements that do appear must also be declared. The first one needed is YEAR. This is the element declaration for the YEAR element:

```
<!ELEMENT YEAR (#PCDATA)>
```

This declaration says that a YEAR may contain only parsed character data, that is, text that's not markup. It may not contain children of its own. Therefore, this YEAR element is valid:

```
<YEAR>1998</YEAR>
```

**These YEAR elements are also valid:**

```
<YEAR>98</YEAR>
<YEAR>1998 C.E.</YEAR>
<YEAR>
  The year of our lord one thousand,
  nine hundred, &amp; ninety-eight
</YEAR>
```

**Even this YEAR element is valid because XML does not attempt to validate the contents of PCDATA, only that it is text that doesn't contain markup.**

```
<YEAR>Delicious, delicious, oh how boring</YEAR>
```

**However, this YEAR element is invalid because it contains child elements:**

```
<YEAR>
  <MONTH>January</MONTH>
  <MONTH>February</MONTH>
  <MONTH>March</MONTH>
  <MONTH>April</MONTH>
  <MONTH>May</MONTH>
  <MONTH>June</MONTH>
  <MONTH>July</MONTH>
  <MONTH>August</MONTH>
  <MONTH>September</MONTH>
  <MONTH>October</MONTH>
  <MONTH>November</MONTH>
  <MONTH>December</MONTH>
</YEAR>
```

**The SEASON and YEAR element declarations are included in the document type declaration, like this:**

```
<!DOCTYPE SEASON [
  <!ELEMENT SEASON ANY>
  <!ELEMENT YEAR (PCDATA)>
]>
```

**As usual, spacing and indentation are not significant. The order in which the element declarations appear isn't relevant either. This next document type declaration means exactly the same thing:**

```
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (PCDATA)>
  <!ELEMENT SEASON ANY>
]>
```

Both of these say that a `SEASON` element may contain parsed character data and any number of any other declared elements in any order. The only other such declared element is `YEAR`, which may contain only parsed character data. For instance, consider the document in Listing 8-5.

### Listing 8-5: A valid document

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT SEASON ANY>
]>
<SEASON>
  <YEAR>1998</YEAR>
</SEASON>
```

Because the `SEASON` element may also contain parsed character data, you can add additional text outside of the `YEAR`. Listing 8-6 demonstrates this.

### Listing 8-6: A valid document that contains a `YEAR` and normal text

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT SEASON ANY>
]>
<SEASON>
  <YEAR>1998</YEAR>
  Major League Baseball
</SEASON>
```

Eventually we'll disallow documents such as this. However, for now it's legal because `SEASON` is declared to accept `ANY` content. Most of the time it's easier to start with `ANY` for an element until you define all of its children. Then you can replace it with the actual children you want to use.

You can attach a simple style sheet, such as the `baseballstats.css` style sheet developed in Chapter 4, to Listing 8-6 — as shown in Listing 8-7 — and load it into a Web browser, as shown in Figure 8-7. The `baseballstats.css` style sheet contains

style rules for elements that aren't present in the DTD or the document part of Listing 8-7, but this is not a problem. Web browsers simply ignore any style rules for elements that aren't present in the document.

### Listing 8-7: A valid document that contains a style sheet, a YEAR, and normal text

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="baseballstats.css"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT SEASON ANY>
]>
<SEASON>
  <YEAR>1998</YEAR>
  Major League Baseball
</SEASON>
```



**Figure 8-7:** A valid document that contains a style sheet, a YEAR element, and normal text displayed in Internet Explorer 5.0

## Child Lists

Because the SEASON element was declared to accept any element as a child, elements could be tossed in willy-nilly. This is useful when you have text that's more or less unstructured, such as a magazine article where paragraphs, sidebars, bulleted lists, numbered lists, graphs, photographs, and subheads may appear pretty much anywhere in the document. However, sometimes you may want to exercise more discipline and control over the placement of your data. For example,

you could require that every `LEAGUE` have one `LEAGUE_NAME`, that every `PLAYER` have a `GIVEN_NAME` and a `SURNAME`, and that the `GIVEN_NAME` come before the `SURNAME`.

To declare that a `LEAGUE` must have a name, simply declare a `LEAGUE_NAME` element, then include `LEAGUE_NAME` in parentheses at the end of the `LEAGUE` declaration, like this:

```
<!ELEMENT LEAGUE (LEAGUE_NAME)>
<!ELEMENT LEAGUE_NAME (#PCDATA)>
```

Each element should be declared in its own `<!ELEMENT>` declaration exactly once, even if it appears as a child in other `<!ELEMENT>` declarations. Here I've placed the declaration `LEAGUE_NAME` after the declaration of `LEAGUE` that refers to it, but that doesn't matter. XML allows these sorts of forward references. The order in which the element tags appear is irrelevant as long as their declarations are all contained inside the DTD.

You can add these two declarations to the document, and then include `LEAGUE` and `LEAGUE_NAME` elements in the `SEASON`. Listing 8-8 demonstrates this. Figure 8-8 shows the rendered document.

### Listing 8-8: A `SEASON` with two `LEAGUE` children

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="baseballstats.css"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT LEAGUE (LEAGUE_NAME)>
  <!ELEMENT LEAGUE_NAME (#PCDATA)>
  <!ELEMENT SEASON ANY>
]>
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>American League</LEAGUE_NAME>
  </LEAGUE>
  <LEAGUE>
    <LEAGUE_NAME>National League</LEAGUE_NAME>
  </LEAGUE>
</SEASON>
```



**Figure 8-8:** A valid document that contains a style sheet, a YEAR element, and two LEAGUE children

## Sequences

Let's restrict the SEASON element as well. A SEASON contains exactly one YEAR, followed by exactly two LEAGUE elements. Instead of saying that a SEASON can contain ANY elements, specify these three children by including them in SEASON's element declaration, enclosed in parentheses and separated by commas, as follows:

```
<!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
```

A list of child elements separated by commas is called a sequence. With this declaration, every valid SEASON element must contain exactly one YEAR element, followed by exactly two LEAGUE elements, and nothing else. The complete document type declaration now looks like this:

```
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT LEAGUE (LEAGUE_NAME)>
  <!ELEMENT LEAGUE_NAME (#PCDATA)>
  <!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
]>
```

The document part of Listing 8-8 does adhere to this DTD because its SEASON element contains one YEAR child followed by two LEAGUE children, and nothing else. However, if the document included only one LEAGUE, then the document, though well-formed, would be invalid. Similarly, if the LEAGUE came before the YEAR element instead of after it, or if the LEAGUE element had YEAR children, or if the document in any other way did not adhere to the DTD, then the document would be invalid and validating parsers would reject it.

It's straightforward to expand these techniques to cover divisions. As well as a LEAGUE\_NAME, each LEAGUE has three DIVISION children. For example:

```
<!ELEMENT LEAGUE (LEAGUE_NAME, DIVISION, DIVISION, DIVISION)>
```

## One or More Children

Each DIVISION has a DIVISION\_NAME and between four and six TEAM children. Specifying the DIVISION\_NAME is easy. This is demonstrated below:

```
<!ELEMENT DIVISION (DIVISION_NAME)>
<!ELEMENT DIVISION_NAME (#PCDATA)>
```

However, the TEAM children are trickier. It's easy to say you want four TEAM children in a DIVISION, as shown below:

```
<!ELEMENT DIVISION (DIVISION_NAME, TEAM, TEAM, TEAM, TEAM)>
```

Five and six are not harder. But how do you say you want between four and six inclusive? In fact, XML doesn't provide an easy way to do this. But you can say you want one or more of a given element by placing a plus sign (+) after the element name in the child list. For example:

```
<!ELEMENT DIVISION (DIVISION_NAME, TEAM+)>
```

This says that a DIVISION element must contain a DIVISION\_NAME element followed by one or more TEAM elements.

 Tip

There is a hard way to say that a DIVISION contains between four and six TEAM elements, but not three and not seven. However, it's so ridiculously complex that nobody would actually use it in practice. Once you finish reading this chapter, see if you can figure out how to do it.

## Zero or More Children

Each TEAM should contain one TEAM\_CITY, one TEAM\_NAME, and an indefinite number of PLAYER elements. In reality, you need at least nine players for a baseball team. However, in the examples in this book, many teams are listed without players for reasons of space. Thus, we want to specify that a TEAM can contain zero or more PLAYER children. Do this by appending an asterisk (\*) to the element name in the child list. For example:

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

## Zero or One Child

The final elements in the document to be brought into play are the children of the `PLAYER`. All of these are simple elements that contain only text. Here are their declarations:

```

<!ELEMENT SURNAME (#PCDATA)>
<!ELEMENT GIVEN_NAME (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT GAMES (#PCDATA)>
<!ELEMENT GAMES_STARTED (#PCDATA)>
<!ELEMENT AT_BATS (#PCDATA)>
<!ELEMENT RUNS (#PCDATA)>
<!ELEMENT HITS (#PCDATA)>
<!ELEMENT DOUBLES (#PCDATA)>
<!ELEMENT TRIPLES (#PCDATA)>
<!ELEMENT HOME_RUNS (#PCDATA)>
<!ELEMENT RBI (#PCDATA)>
<!ELEMENT STEALS (#PCDATA)>
<!ELEMENT CAUGHT_STEALING (#PCDATA)>
<!ELEMENT SACRIFICE_HITS (#PCDATA)>
<!ELEMENT SACRIFICE_FLIES (#PCDATA)>
<!ELEMENT ERRORS (#PCDATA)>
<!ELEMENT WALKS (#PCDATA)>
<!ELEMENT STRUCK_OUT (#PCDATA)>
<!ELEMENT HIT_BY_PITCH (#PCDATA)>
<!ELEMENT COMPLETE_GAMES (#PCDATA)>
<!ELEMENT SHUT_OUTS (#PCDATA)>
<!ELEMENT ERA (#PCDATA)>
<!ELEMENT INNINGS (#PCDATA)>
<!ELEMENT EARNED_RUNS (#PCDATA)>
<!ELEMENT HIT_BATTER (#PCDATA)>
<!ELEMENT WILD_PITCHES (#PCDATA)>
<!ELEMENT BALK (#PCDATA)>
<!ELEMENT WALKED_BATTER (#PCDATA)>
<!ELEMENT WINS (#PCDATA)>
<!ELEMENT LOSSES (#PCDATA)>
<!ELEMENT SAVES (#PCDATA)>
<!ELEMENT COMPLETE_GAMES (#PCDATA)>
<!ELEMENT STRUCK_OUT_BATTER (#PCDATA)>

```

Now we can write the declaration for the `PLAYER` element. All players have one `SURNAME`, one `GIVEN_NAME`, one `POSITION`, and one `GAMES`. We could declare that each `PLAYER` also has one `AT_BATS`, `RUNS`, `HITS`, and so forth. However, I'm not sure it's accurate to list zero runs for a pitcher who hasn't batted. For one thing, this likely will lead to division by zero errors when you start calculating batting averages and so on. If a particular element doesn't apply to a given player, or if it's not available, then the more sensible thing to do is to omit the particular statistic from the player's information. We don't allow more than one of each element for a given

player. Thus, we want zero or one element of the given type. Indicate this in a child element list by appending a question mark (?) to the element, as shown below:

```
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, POSITION, GAMES,
  GAMES_STARTED, AT_BATS?, RUNS?, HITS?, DOUBLES?,
  TRIPLES?, HOME_RUNS?, RBI?, STEALS?, CAUGHT_STEALING?,
  SACRIFICE_HITS?, SACRIFICE_FLIES?, ERRORS?, WALKS?,
  STRUCK_OUT?, HIT_BY_PITCH?, WINS?, LOSSES?, SAVES?,
  COMPLETE_GAMES?, SHUT_OUTS?, ERA?, INNINGS?, EARNED_RUNS?,
  HIT_BATTER?, WILD_PITCHES?, BALK?, WALKED_BATTER?,
  STRUCK_OUT_BATTER?)
>
```

This says that every `PLAYER` has a `SURNAME`, `GIVEN_NAME`, `POSITION`, `GAMES`, and `GAMES_STARTED`. Furthermore, each player may or may not have a single `AT_BATS`, `RUNS`, `HITS`, `DOUBLES`, `TRIPLES`, `HOME_RUNS`, `RBI`, `STEALS`, `CAUGHT_STEALING`, `SACRIFICE_HITS`, `SACRIFICE_FLIES`, `ERRORS`, `WALKS`, `STRUCK_OUT`, and `HIT_BY_PITCH`.

## The Complete Document and DTD

We now have a complete DTD for baseball statistics. This DTD, along with the document part of Listing 8-4, is shown in Listing 8-9.



Listing 8-9 only covers a single team and nine players. On the CD-ROM you'll find a document containing statistics for all 1998 Major League teams and players in the `examples/baseball/1998validstats.xml` directory.

### Listing 8-9: A valid XML document on baseball statistics with a DTD

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT LEAGUE (LEAGUE_NAME, DIVISION, DIVISION, DIVISION)>
  <!ELEMENT LEAGUE_NAME (#PCDATA)>
  <!ELEMENT DIVISION_NAME (#PCDATA)>
  <!ELEMENT DIVISION (DIVISION_NAME, TEAM+)>
  <!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
  <!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
  <!ELEMENT TEAM_CITY (#PCDATA)>
  <!ELEMENT TEAM_NAME (#PCDATA)>
  <!ELEMENT PLAYER (GIVEN_NAME, SURNAME, POSITION, GAMES,
    GAMES_STARTED, WINS?, LOSSES?, SAVES?,
```

*Continued*

## Listing 8-9 (continued)

```

    AT_BATS?, RUNS?, HITS?, DOUBLES?, TRIPLES?, HOME_RUNS?,
    RBI?, STEALS?, CAUGHT_STEALING?, SACRIFICE_HITS?,
    SACRIFICE_FLIES?, ERRORS?, WALKS?, STRUCK_OUT?,
    HIT_BY_PITCH?, COMPLETE_GAMES?, SHUT_OUTS?, ERA?, INNINGS?,
    EARNED_RUNS?, HIT_BATTER?, WILD_PITCHES?, BALK?,
    WALKED_BATTER?, STRUCK_OUT_BATTER?)
>
<!ELEMENT SURNAME (#PCDATA)>
<!ELEMENT GIVEN_NAME (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT GAMES (#PCDATA)>
<!ELEMENT GAMES_STARTED (#PCDATA)>
<!ELEMENT COMPLETE_GAMES (#PCDATA)>
<!ELEMENT WINS (#PCDATA)>
<!ELEMENT LOSSES (#PCDATA)>
<!ELEMENT SAVES (#PCDATA)>
<!ELEMENT AT_BATS (#PCDATA)>
<!ELEMENT RUNS (#PCDATA)>
<!ELEMENT HITS (#PCDATA)>
<!ELEMENT DOUBLES (#PCDATA)>
<!ELEMENT TRIPLES (#PCDATA)>
<!ELEMENT HOME_RUNS (#PCDATA)>
<!ELEMENT RBI (#PCDATA)>
<!ELEMENT STEALS (#PCDATA)>
<!ELEMENT CAUGHT_STEALING (#PCDATA)>
<!ELEMENT SACRIFICE_HITS (#PCDATA)>
<!ELEMENT SACRIFICE_FLIES (#PCDATA)>
<!ELEMENT ERRORS (#PCDATA)>
<!ELEMENT WALKS (#PCDATA)>
<!ELEMENT STRUCK_OUT (#PCDATA)>
<!ELEMENT HIT_BY_PITCH (#PCDATA)>
<!ELEMENT SHUT_OUTS (#PCDATA)>
<!ELEMENT ERA (#PCDATA)>
<!ELEMENT INNINGS (#PCDATA)>
<!ELEMENT HOME_RUNS_AGAINST (#PCDATA)>
<!ELEMENT RUNS_AGAINST (#PCDATA)>
<!ELEMENT EARNED_RUNS (#PCDATA)>
<!ELEMENT HIT_BATTER (#PCDATA)>
<!ELEMENT WILD_PITCHES (#PCDATA)>
<!ELEMENT BALK (#PCDATA)>
<!ELEMENT WALKED_BATTER (#PCDATA)>
<!ELEMENT STRUCK_OUT_BATTER (#PCDATA)>
]
<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>
    <PLAYER>
      <GIVEN_NAME>Eric</GIVEN_NAME>
      <SURNAME>Ludwick</SURNAME>
      <POSITION>Starting Pitcher</POSITION>
      <GAMES>13</GAMES>
      <GAMES_STARTED>6</GAMES_STARTED>
      <WINS>1</WINS>
      <LOSSES>4</LOSSES>
      <SAVES>0</SAVES>
      <COMPLETE_GAMES>0</COMPLETE_GAMES>
      <SHUT_OUTS>0</SHUT_OUTS>
      <ERA>7.44</ERA>
      <INNINGS>32.2</INNINGS>
      <EARNED_RUNS>31</EARNED_RUNS>
      <HIT_BATTER>27</HIT_BATTER>
      <WILD_PITCHES>0</WILD_PITCHES>
      <BALK>2</BALK>
      <WALKED_BATTER>0</WALKED_BATTER>
      <STRUCK_OUT_BATTER>17</STRUCK_OUT_BATTER>
    </PLAYER>
    <PLAYER>
      <GIVEN_NAME>Brian</GIVEN_NAME>
      <SURNAME>Daubach</SURNAME>
      <POSITION>First Base</POSITION>
      <GAMES>10</GAMES>
      <GAMES_STARTED>3</GAMES_STARTED>
      <AT_BATS>15</AT_BATS>
      <RUNS>0</RUNS>
      <HITS>3</HITS>
      <DOUBLES>1</DOUBLES>
      <TRIPLES>0</TRIPLES>
      <HOME_RUNS>0</HOME_RUNS>
      <RBI>3</RBI>
      <STEALS>0</STEALS>
      <CAUGHT_STEALING>0</CAUGHT_STEALING>
      <SACRIFICE_HITS>0</SACRIFICE_HITS>
      <SACRIFICE_FLIES>0</SACRIFICE_FLIES>
      <ERRORS>0</ERRORS>
      <WALKS>1</WALKS>
      <STRUCK_OUT>5</STRUCK_OUT>
      <HIT_BY_PITCH>1</HIT_BY_PITCH>
    </PLAYER>
  </TEAM>
</TEAM>
<TEAM>
  <TEAM_CITY>Montreal</TEAM_CITY>
  <TEAM_NAME>Expos</TEAM_NAME>

```

```

    </TEAM>
    <TEAM>
      <TEAM_CITY>New York</TEAM_CITY>
      <TEAM_NAME>Mets</TEAM_NAME>
    </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>
    </TEAM>
  </DIVISION>
  <DIVISION>
    <DIVISION_NAME>West</DIVISION_NAME>
    <TEAM>
      <TEAM_CITY>Anaheim</TEAM_CITY>
      <TEAM_NAME>Angels</TEAM_NAME>
    </TEAM>
  </DIVISION>
</LEAGUE>
</SEASON>

```

---

Listing 8-9 is not the only possible document that matches this DTD, however. Listing 8-10 is also a valid document, because it contains all required elements in their required order and does not contain any elements that aren't declared. This is probably the smallest reasonable document that you can create that fits the DTD. The limiting factors are the requirements that each SEASON contain two LEAGUE children, that each LEAGUE contain three DIVISION children, and that each DIVISION contain at least one TEAM.

### Listing 8-10: Another XML document that's valid according to the baseball DTD

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON [
  <!ELEMENT YEAR (#PCDATA)>
  <!ELEMENT LEAGUE (LEAGUE_NAME, DIVISION, DIVISION, DIVISION)>
  <!ELEMENT LEAGUE_NAME (#PCDATA)>
  <!ELEMENT DIVISION_NAME (#PCDATA)>
  <!ELEMENT DIVISION (DIVISION_NAME, TEAM+)>
  <!ELEMENT SEASON (YEAR, LEAGUE, LEAGUE)>
  <!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
  <!ELEMENT TEAM_CITY (#PCDATA)>
  <!ELEMENT TEAM_NAME (#PCDATA)>
  <!ELEMENT PLAYER (GIVEN_NAME, SURNAME, POSITION, GAMES,
    GAMES_STARTED, COMPLETE_GAMES?, WINS?, LOSSES?, SAVES?,
    AT_BATS?, RUNS?, HITS?, DOUBLES?, TRIPLES?, HOME_RUNS?,
    RBI?, STEALS?, CAUGHT_STEALING?, SACRIFICE_HITS?,
    SACRIFICE_FLIES?, ERRORS?, WALKS?, STRUCK_OUT?,
    HIT_BY_PITCH?, COMPLETE_GAMES?, SHUT_OUTS?, ERA?, INNINGS?,
    EARNED_RUNS?, HIT_BATTER?, WILD_PITCHES?, BALK?,
    WALKED_BATTER?, STRUCK_OUT_BATTER?)
  >
  <!ELEMENT SURNAME (#PCDATA)>
  <!ELEMENT GIVEN_NAME (#PCDATA)>
  <!ELEMENT POSITION (#PCDATA)>
  <!ELEMENT GAMES (#PCDATA)>
  <!ELEMENT GAMES_STARTED (#PCDATA)>
  <!ELEMENT COMPLETE_GAMES (#PCDATA)>
  <!ELEMENT WINS (#PCDATA)>
  <!ELEMENT LOSSES (#PCDATA)>
  <!ELEMENT SAVES (#PCDATA)>
  <!ELEMENT AT_BATS (#PCDATA)>
  <!ELEMENT RUNS (#PCDATA)>
  <!ELEMENT HITS (#PCDATA)>
  <!ELEMENT DOUBLES (#PCDATA)>
  <!ELEMENT TRIPLES (#PCDATA)>
  <!ELEMENT HOME_RUNS (#PCDATA)>
```

*Continued*

## Listing 8-10 (continued)

```

<!ELEMENT RBI (#PCDATA)>
<!ELEMENT STEALS (#PCDATA)>
<!ELEMENT CAUGHT_STEALING (#PCDATA)>
<!ELEMENT SACRIFICE_HITS (#PCDATA)>
<!ELEMENT SACRIFICE_FLIES (#PCDATA)>
<!ELEMENT ERRORS (#PCDATA)>
<!ELEMENT WALKS (#PCDATA)>
<!ELEMENT STRUCK_OUT (#PCDATA)>
<!ELEMENT HIT_BY_PITCH (#PCDATA)>
<!ELEMENT SHUT_OUTS (#PCDATA)>
<!ELEMENT ERA (#PCDATA)>
<!ELEMENT INNINGS (#PCDATA)>
<!ELEMENT HOME_RUNS_AGAINST (#PCDATA)>
<!ELEMENT RUNS_AGAINST (#PCDATA)>
<!ELEMENT EARNED_RUNS (#PCDATA)>
<!ELEMENT HIT_BATTER (#PCDATA)>
<!ELEMENT WILD_PITCHES (#PCDATA)>
<!ELEMENT BALK (#PCDATA)>
<!ELEMENT WALKED_BATTER (#PCDATA)>
<!ELEMENT STRUCK_OUT_BATTER (#PCDATA)>

]>
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>National</LEAGUE_NAME>
    <DIVISION>
      <DIVISION_NAME>East</DIVISION_NAME>
      <TEAM>
        <TEAM_CITY>Atlanta</TEAM_CITY>
        <TEAM_NAME>Braves</TEAM_NAME>
      </TEAM>
      <TEAM>
        <TEAM_CITY>Florida</TEAM_CITY>
        <TEAM_NAME>Marlins</TEAM_NAME>
      </TEAM>
      <TEAM>
        <TEAM_CITY>Montreal</TEAM_CITY>
        <TEAM_NAME>Expos</TEAM_NAME>
      </TEAM>
      <TEAM>
        <TEAM_CITY>New York</TEAM_CITY>
        <TEAM_NAME>Mets</TEAM_NAME>
      </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>
    </TEAM>
  </DIVISION>
  <DIVISION>
    <DIVISION_NAME>West</DIVISION_NAME>
    <TEAM>
      <TEAM_CITY>Anaheim</TEAM_CITY>
      <TEAM_NAME>Angels</TEAM_NAME>
    </TEAM>
  </DIVISION>
</LEAGUE>
</SEASON>
```

---

## Choices

In general, a single parent element has many children. To indicate that the children must occur in sequence, they are separated by commas. However, each such child element may be suffixed with a question mark, a plus sign, or an asterisk to adjust the number of times it appears in that place in the sequence.

So far, the assumption has been made that child elements appear or do not appear in a specific order. You may, however, wish to make your DTD more flexible, such as by allowing document authors to choose between different elements in a given place. For example, in a DTD describing a purchase by a customer, each `PAYMENT` element might have either a `CREDIT_CARD` child or a `CASH` child providing information about the method of payment. However, an individual `PAYMENT` would not have both.

You can indicate that the document author needs to input either one or another element by separating child elements with a vertical bar (`|`) rather than a comma (`,`) in the parent's element declaration. For example, the following says that the `PAYMENT` element must have a single child of type `CASH` or `CREDIT_CARD`.

```
<!ELEMENT PAYMENT (CASH | CREDIT_CARD)>
```

This sort of content specification is called a choice. You can separate any number of children with vertical bars when you want exactly one of them to be used. For example, the following says that the `PAYMENT` element must have a single child of type `CASH`, `CREDIT_CARD`, or `CHECK`.

```
<!ELEMENT PAYMENT (CASH | CREDIT_CARD | CHECK)>
```

The vertical bar is even more useful when you group elements with parentheses. You can group combinations of elements in parentheses, then suffix the parentheses with asterisks, question marks, and plus signs to indicate that particular combinations of elements must occur zero or more, zero or one, or one or more times.

## Children with Parentheses

The final thing you need to know about arranging child elements in parent element declarations is how to group elements with parentheses. Each set of parentheses combines several elements as a single element. This parenthesized element can then be nested inside other parentheses in place of a single element. Furthermore, it may then have a plus sign, a comma, or a question mark affixed to it. You can group these parenthesized combinations into still larger parenthesized groups to produce quite complex structures. This is a very powerful technique.

For example, consider a list composed of two elements that must alternate with each other. This is essentially how HTML's definition list works. Each `<dt>` tag should match one `<dd>` tag. If you replicate this structure in XML, the declaration of the `d1` element looks like this:

```
<!ELEMENT d1 (dt, dd)*>
```

The parentheses indicate that it's the matched `<dt><dd>` pair being repeated, not `<dd>` alone.

Often elements appear in more or less random orders. News magazine articles generally have a title mostly followed by paragraphs of text, but with graphs, photos, sidebars, subheads, and pull quotes interspersed throughout, perhaps with a byline at the end. You can indicate this sort of arrangement by listing all the possible child elements in the parent's element declaration separated by vertical bars and grouped inside parentheses. You can then place an asterisk outside the closing parenthesis to indicate that zero or more occurrences of any of the elements in the parentheses are allowed. For example;

```
<!ELEMENT ARTICLE (TITLE, (P | PHOTO | GRAPH | SIDEBAR
| PULLQUOTE | SUBHEAD)*, BYLINE?)>
```

As another example, suppose you want to say that a DOCUMENT element, rather than having any children at all, must have one TITLE followed by any number of paragraphs of text and images that may be freely intermingled, followed by an optional SIGNATURE block. Write its element declaration this way:

```
<!ELEMENT DOCUMENT (TITLE, (PARAGRAPH | IMAGE)*, SIGNATURE?)>
```

This is not the only way to describe this structure. In fact, it may not even be the best way. An alternative is to declare a BODY element that contains PARAGRAPH and IMAGE elements and nest that between the TITLE and the SIGNATURE. For example:

```
<!ELEMENT DOCUMENT (TITLE, BODY, SIGNATURE?)>
<!ELEMENT BODY ((PARAGRAPH | IMAGE)*)>
```

The difference between these two approaches is that the second requires an additional BODY element in the document. This element provides an additional level of organization that may (or may not) be useful to the application that's reading the document. The question to ask is whether the reader of this document (who may be another computer program) may want to consider the BODY as a single item in its own right, separate from the TITLE and the SIGNATURE and distinguished from the sum of its elements.

For another example, consider international addresses. Addresses outside the United States don't always follow U.S. conventions. In particular, postal codes sometimes precede the state or follow the country, as in these two examples:

Doberman-YPPAN  
Box 2021  
St. Nicholas QUEBEC  
CAN GOS-3LO

*or*

Editions Sybex  
10/12 Villa Coeur-de-Vey  
75685 Paris Cedex 14  
France

Although your mail will probably arrive even if pieces of the address are out of order, it's better to allow an address to be more flexible. Here's one address element declaration that permits this:

```
<!ELEMENT ADDRESS (STREET+, (CITY | STATE | POSTAL_CODE
| COUNTRY)*)>
```

This says that an ADDRESS element must have one or more STREET children followed by any number of CITY, STATE, POSTAL\_CODE, or COUNTRY elements. Even this is less than ideal if you'd like to allow for no more than one of each. Unfortunately, this is beyond the power of a DTD to enforce. By allowing a more flexible ordering of elements, you give up some ability to control the maximum number of each element.

On the other hand, you may have a list composed of different kinds of elements, which may appear in an arbitrary order, as in a list of recordings that may contain CDs, albums, and tapes. An element declaration to differentiate between the different categories for this list would look like this:

```
<!ELEMENT MUSIC_LIST (CD | ALBUM | TAPE)*>
```

You could use parentheses in the baseball DTD to specify different sets of statistics for pitchers and batters. Each player could have one set or the other, but not both. The element declaration looks like this:

```
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, POSITION, GAMES,
GAMES_STARTED, (( COMPLETE_GAMES?, WINS?, LOSSES?, SAVES?,
SHUT_OUTS?, ERA?, INNINGS?, EARNED_RUNS?, HIT_BATTER?,
WILD_PITCHES?, BALK?, WALKED_BATTER?, STRUCK_OUT_BATTER? )
|(AT_BATS?, RUNS?, HITS?, DOUBLES?, TRIPLES?, HOME_RUNS?,
RBI?, STEALS?, CAUGHT_STEALING?, SACRIFICE_HITS?,
SACRIFICE_FLIES?, ERRORS?, WALKS?, STRUCK_OUT?,
HIT_BY_PITCH? )))>
```

There are still a few things that are difficult to handle in element declarations. For example, there's no good way to say that a document must begin with a TITLE element and end with a SIGNATURE element, but may contain any other elements between those two. This is because ANY may not combine with other child elements.

And, in general, the less precise you are about where things appear, the less control you have over how many of them there are. For example, you can't say that a document should have exactly one TITLE element but that the TITLE may appear anywhere in the document.

Nonetheless, using parentheses to create blocks of elements, either in sequence with a comma or in parallel with a vertical bar, enables you to create complicated

structures with detailed rules for how different elements follow one another. Try not to go overboard with this though. Simpler solutions are better. The more complicated your DTD is, the harder it is to write valid files that satisfy the DTD, to say nothing of the complexity of maintaining the DTD itself.

## Mixed Content

You may have noticed that in most of the examples shown so far, elements either contained child elements or parsed character data, but not both. The only exceptions were the root elements in early examples where the full list of tags was not yet developed. In these cases, because the root element could contain ANY data, it was allowed to contain both child elements and raw text.

You can declare tags that contain both child elements and parsed character data. This is called *mixed content*. You can use this to allow an arbitrary block of text to be suffixed to each TEAM. For example:

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

Mixing child elements with parsed character data severely restricts the structure you can impose on your documents. In particular, you can specify only the names of the child elements that can appear. You cannot constrain the order in which they appear, the number of each that appears, or whether they appear at all. In terms of DTDs, think of this as meaning that the child part of the DTD must look like this:

```
<!ELEMENT PARENT (#PCDATA | CHILD1 | CHILD2 | CHILD3 )* >
```

Almost everything else, other than changing the number of children, is invalid. You cannot use commas, question marks, or plus signs in an element declaration that includes #PCDATA. A list of elements and #PCDATA separated by vertical bars is valid. Any other use is not. For example, the following is illegal:

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

The primary reason to mix content is when you're in the process of converting old text data to XML, and testing your DTD by validating as you add new tags rather than finishing the entire conversion and then trying to find the bugs. This is a good technique, and I do recommend you use it — after all, it is much easier to recognize a mistake in your code immediately after you made it rather than several hours later — however, this is only a crutch for use when developing. It should not be visible to the end-user. When your DTD is finished it should not mix element children with parsed character data. You can always create a new tag that holds parsed character data.

For example, you can include a block of text at the end of each `TEAM` element by declaring a new `BLURB` that holds only `#PCDATA` and adding it as the last child element of `TEAM`. Here's how this looks:

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

This does not significantly change the text of the document. All it does is add one more optional element with its opening and closing tags to each `TEAM` element. However, it does make the document much more robust. Furthermore, XML applications that receive the tree from the XML processor have an easier time handling the data when it's in the more structured format allowed by nonmixed content.

## Empty Elements

As discussed earlier, it's occasionally useful to define an element that has no content. Examples in HTML include the image, horizontal rule, and break `<IMG>`, `<HR>`, and `<BR>`. In XML, such empty elements are identified by empty tags that end with `/>`, such as `<IMG/>`, `<HR/>`, and `<BR/>`.

Valid documents must declare both the empty and nonempty elements used. Because empty elements by definition don't have children, they're easy to declare. Use an `<!ELEMENT>` declaration containing the name of the empty element as normal, but use the keyword `EMPTY` (case-sensitive as all XML tags are) instead of a list of children. For example:

```
<!ELEMENT BR EMPTY>
<!ELEMENT IMG EMPTY>
<!ELEMENT HR EMPTY>
```

Listing 8-11 is a valid document that uses both empty and nonempty elements.

### Listing 8-11: A valid document that uses empty tags

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT (TITLE, SIGNATURE)>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT COPYRIGHT (#PCDATA)>
  <!ELEMENT EMAIL (#PCDATA)>
  <!ELEMENT BR EMPTY>
  <!ELEMENT HR EMPTY>
  <!ELEMENT LAST_MODIFIED (#PCDATA)>
  <!ELEMENT SIGNATURE (HR, COPYRIGHT, BR, EMAIL,
```

```

        BR, LAST_MODIFIED)>
    ]>
<DOCUMENT>
  <TITLE>Empty Tags</TITLE>
  <SIGNATURE>
    <HR/>
    <COPYRIGHT>1998 Elliotte Rusty Harold</COPYRIGHT><BR/>
    <EMAIL>elharo@metalab.unc.edu</EMAIL><BR/>
    <LAST_MODIFIED>Thursday, April 22, 1999</LAST_MODIFIED>
  </SIGNATURE>
</DOCUMENT>

```

---

## Comments in DTDs

DTDs can contain comments, just like the rest of an XML document. These comments cannot appear inside a declaration, but they can appear outside one. Comments are often used to organize the DTD in different parts, to document the allowed content of particular elements, and to further explain what an element is. For example, the element declaration for the `YEAR` element might have a comment such as this:

```

<!-- A four digit year like 1998, 1999, or 2000 -->
<!ELEMENT YEAR (#PCDATA)>

```

As with all comments, this is only for the benefit of people reading the source code. XML processors will ignore it.

One possible use of comments is to define abbreviations used in the markup. For example, in this and previous chapters, I've avoided using abbreviations for baseball terms because they're simply not obvious to the casual fan. An alternative approach is to use abbreviations but define them with comments in the DTD. Listing 8-12 is similar to previous baseball examples, but uses DTD comments and abbreviated tags.

### Listing 8-12: A valid XML document that uses abbreviated tags defined in DTD comments

```

<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON [
    <!ELEMENT YEAR (#PCDATA)>

```

*Continued*

## Listing 8-12 (continued)

```

<!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)>
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, P, G,
  GS, AB?, R?, H?, D?, T?, HR?, RBI?, SB?, CS?,
  SH?, SF?, E?, BB?, S?, HBP?, CG?, SO?, ERA?, IP?,
  HRA?, RA?, ER?, HB?, WP?, B?, WB?, K?)
>

<!-- ===== -->
<!-- Player Info -->
<!-- Player's last name -->
<!ELEMENT SURNAME (#PCDATA)>

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

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

<!-- Games Played -->
<!ELEMENT G (#PCDATA)>

<!-- 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)>

<!-- Shut Outs -->
<!ELEMENT SO (#PCDATA)>

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

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

*Continued*

## Listing 8-12 (continued)

```

    <!-- 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 -->

]>
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>National</LEAGUE_NAME>
    <DIVISION>
      <DIVISION_NAME>East</DIVISION_NAME>
      <TEAM>
        <TEAM_CITY>Atlanta</TEAM_CITY>
        <TEAM_NAME>Braves</TEAM_NAME>
        <PLAYER>
          <GIVEN_NAME>Ozzie</GIVEN_NAME>
          <SURNAME>Guillen</SURNAME>
          <P>Shortstop</P>
          <G>83</G>
          <GS>59</GS>
          <AB>264</AB>
          <R>35</R>
          <H>73</H>

```

```

        <D>15</D>
        <T>1</T>
        <HR>1</HR>
        <RBI>22</RBI>
        <SB>1</SB>
        <CS>4</CS>
        <SH>4</SH>
        <SF>2</SF>
        <E>6</E>
        <BB>24</BB>
        <S>25</S>
        <HBP>1</HBP>
    </PLAYER>
</TEAM>
<TEAM>
    <TEAM_CITY>Florida</TEAM_CITY>
    <TEAM_NAME>Marlins</TEAM_NAME>
</TEAM>
<TEAM>
    <TEAM_CITY>Montreal</TEAM_CITY>
    <TEAM_NAME>Expos</TEAM_NAME>
</TEAM>
<TEAM>
    <TEAM_CITY>New York</TEAM_CITY>
    <TEAM_NAME>Mets</TEAM_NAME>
</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>

```

*Continued*

## Listing 8-12 (continued)

```
<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>
  </TEAM>
</DIVISION>
<DIVISION>
  <DIVISION_NAME>West</DIVISION_NAME>
  <TEAM>
    <TEAM_CITY>Anaheim</TEAM_CITY>
    <TEAM_NAME>Angels</TEAM_NAME>
  </TEAM>
</DIVISION>
</LEAGUE>
</SEASON>
```

---

When the entire Major League is included, the resulting document shrinks from 699K with long tags to 391K with short tags, a savings of 44 percent. The information content, however, is virtually the same. Consequently, the compressed sizes of the two documents are much closer, 58K for the document with short tags versus 66K for the document with long tags.

There's no limit to the amount of information you can or should include in comments. Including more does make your DTDs longer (and thus both harder to scan and slower to download). However, in the next couple of chapters, you'll learn ways to reuse the same DTD in multiple XML documents, as well as break long DTDs into more manageable pieces. Thus, the disadvantages of using comments are temporary. I recommend using comments liberally in all of your DTDs, but especially in those intended for public use.

## Sharing Common DTDs Among Documents

Previous valid examples included the DTD in the document's prolog. The real power of XML, however, comes from common DTDs that can be shared among

many documents written by different people. If the DTD is not directly included in the document but is linked in from an external source, changes made to the DTD automatically propagate to all documents using that DTD. On the other hand, backward compatibility is not guaranteed when a DTD is modified. Incompatible changes can break documents.

When you use an external DTD, the document type declaration changes. Instead of including the DTD in square brackets, the `SYSTEM` keyword is followed by an absolute or relative URL where the DTD can be found. For example:

```
<!DOCTYPE root_element_name SYSTEM "DTD_URL">
```

Here `root_element_name` is simply the name of the root element as before, `SYSTEM` is an XML keyword, and `DTD_URL` is a relative or an absolute URL where the DTD can be found. For example:

```
<!DOCTYPE SEASON SYSTEM "baseball.dtd">
```

Let's convert a familiar example to demonstrate this process. Listing 8-12 includes an internal DTD for baseball statistics. We'll convert this listing to use an external DTD. First, strip out the DTD and put it in a file of its own. This is everything between the opening `<!DOCTYPE SEASON [ and the closing ]>` exclusive. `<!DOCTYPE SEASON [ and ]>` are not included. This can be saved in a file called `baseball.dtd`, as shown in Listing 8-13. The file name is not important, though the extension `.dtd` is conventional.

### Listing 8-13: The baseball DTD file

```
<!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)>
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, P, G,
  GS, AB?, R?, H?, D?, T?, HR?, RBI?, SB?, CS?,
  SH?, SF?, E?, BB?, S?, HBP?, CG?, SO?, ERA?, IP?,
```

*Continued*

## Listing 8-13 (continued)

```

    HRA?, RA?, ER?, HB?, WP?, B?, WB?, K?)
  >
<!-- ===== -->
<!-- Player Info -->
<!-- Player's last name -->
<!ELEMENT SURNAME (#PCDATA)>

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

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

<!--Games Played -->
<!ELEMENT G (#PCDATA)>

<!--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)>

<!-- Shut Outs -->
<!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)>
```

*Continued*

## Listing 8-13 (continued)

```

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

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

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

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

```

Next, you need to modify the document itself. The XML declaration is no longer a stand-alone document because it depends on a DTD in another file. Therefore, the `standalone` attribute must be changed to `no`, as follows:

```
<?xml version="1.0" standalone="no"?>
```

Then you must change the `<!DOCTYPE>` tag so it points to the DTD by including the `SYSTEM` keyword and a URL (usually relative) where the DTD is found:

```
<!DOCTYPE SEASON SYSTEM "baseball.dtd">
```

The rest of the document is the same as before. However, now the prolog contains only the XML declaration and the document type declaration. It does not contain the DTD. Listing 8-14 shows the code.

## Listing 8-14: Baseball statistics with an external DTD

```

<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON SYSTEM "baseball.dtd">
<SEASON>
  <YEAR>1998</YEAR>
  <LEAGUE>
    <LEAGUE_NAME>National</LEAGUE_NAME>
    <DIVISION>
      <DIVISION_NAME>East</DIVISION_NAME>
    <TEAM>
      <TEAM_CITY>Atlanta</TEAM_CITY>
      <TEAM_NAME>Braves</TEAM_NAME>
    <PLAYER>

```

```

    <GIVEN_NAME>Ozzie</GIVEN_NAME>
    <SURNAME>Guillen</SURNAME>
    <P>Shortstop</P>
    <G>83</G>
    <GS>59</GS>
    <AB>264</AB>
    <R>35</R>
    <H>73</H>
    <D>15</D>
    <T>1</T>
    <HR>1</HR>
    <RBI>22</RBI>
    <SB>1</SB>
    <CS>4</CS>
    <SH>4</SH>
    <SF>2</SF>
    <E>6</E>
    <BB>24</BB>
    <S>25</S>
    <HBP>1</HBP>
  </PLAYER>
</TEAM>
<TEAM>
  <TEAM_CITY>Florida</TEAM_CITY>
  <TEAM_NAME>Marlins</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>Montreal</TEAM_CITY>
  <TEAM_NAME>Expos</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>New York</TEAM_CITY>
  <TEAM_NAME>Mets</TEAM_NAME>
</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>

```

*Continued*

## Listing 8-14 (continued)

```

        <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>
        </TEAM>
    </DIVISION>
    <DIVISION>
        <DIVISION_NAME>West</DIVISION_NAME>
        <TEAM>
            <TEAM_CITY>Anaheim</TEAM_CITY>
            <TEAM_NAME>Angels</TEAM_NAME>
        </TEAM>
    </DIVISION>
</LEAGUE>
</SEASON>

```

Make sure that both Listing 8-14 and `baseball.dtd` are in the same directory and then load Listing 8-14 into your Web browser as usual. If all is well, you see the same output as when you loaded Listing 8-12. You can now use this same DTD to describe other documents, such as statistics from other years.

Once you add a style sheet, you have the three essential parts of the document stored in three different files. The data is in the document file, the structure and semantics applied to the data is in the DTD file, and the formatting is in the style sheet. This structure enables you to inspect or change any or all of these relatively independently.

The DTD and the document are more closely linked than the document and the style sheet. Changing the DTD generally requires revalidating the document and

may require edits to the document to bring it back into conformance with the DTD. The necessity of this sequence depends on your edits; adding elements is rarely an issue, though removing elements may be problematic.

## DTDs at Remote URLs

If a DTD is applied to multiple documents, you cannot always put the DTD in the same directory as each document for which it is used. Instead, you can use a URL to specify precisely where the DTD is found. For example, let's suppose the baseball DTD is found at `http://metalab.unc.edu/xml/dtds/baseball.dtd`. You can link to it by using the following `<!DOCTYPE>` tag in the prolog:

```
<!DOCTYPE SEASON SYSTEM
    "http://metalab.unc.edu/xml/dtds/baseball.dtd">
```

This example uses a full URL valid from anywhere. You may also wish to locate DTDs relative to the Web server's document root or the current directory. In general, any reference that forms a valid URL relative to the location of the document is acceptable. For example, these are all valid document type declarations:

```
<!DOCTYPE SEASON SYSTEM "/xml/dtds/baseball.dtd">
<!DOCTYPE SEASON SYSTEM "dtds/baseball.dtd">
<!DOCTYPE SEASON SYSTEM "../baseball.dtd">
```

**Note**

A document can't have more than one document type declaration, that is, more than one `<!DOCTYPE>` tag. To use elements declared in more than one DTD, you need to use external parameter entity references. These are discussed in the next chapter.

## Public DTDs

The `SYSTEM` keyword is intended for private DTDs used by a single author or group. Part of the promise of XML, however, is that broader organizations covering an entire industry, such as the ISO or the IEEE, can standardize public DTDs to cover their fields. This standardization saves people from having to reinvent tag sets for the same items and makes it easier for users to exchange interoperable documents.

DTDs designed for writers outside the creating organization use the `PUBLIC` keyword instead of the `SYSTEM` keyword. Furthermore, the DTD gets a name. The syntax follows:

```
<!DOCTYPE root_element_name PUBLIC "DTD_name" "DTD_URL">
```

Once again, *root\_element\_name* is the name of the root element. PUBLIC is an XML keyword indicating that this DTD is intended for broad use and has a name. *DTD\_name* is the name associated with this DTD. Some XML processors may attempt to use this name to retrieve the DTD from a central repository. Finally, *DTD\_URL* is a relative or absolute URL where the DTD can be found if it cannot be retrieved by name from a well-known repository.

DTD names are slightly different from XML names. They may contain only the ASCII alphanumeric characters, the space, the carriage return, the linefeed characters, and the following punctuation marks: - ' () +, / : = ? ; ! \* # @ \$ \_ %. Furthermore, the names of public DTDs follow a few conventions.

If a DTD is an ISO standard, its name begins with the string “ISO.” If a non-ISO standards body has approved the DTD, its name begins with a plus sign (+). If no standards body has approved the DTD, its name begins with a hyphen (-). These initial strings are followed by a double slash (//) and the name of the DTD’s owner, which is followed by another double slash and the type of document the DTD describes. Then there’s another double slash followed by an ISO 639 language identifier, such as EN for English. A complete list of ISO 639 identifiers is available from <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>. For example, the baseball DTD can be named as follows:

```
-//Elliott Rusty Harold//DTD baseball statistics//EN
```

This example says this DTD is not standards-body approved (-), belongs to Elliott Rusty Harold, describes baseball statistics, and is written in English. A full document type declaration pointing to this DTD with this name follows:

```
<!DOCTYPE SEASON PUBLIC
  "-//Elliott Rusty Harold//DTD baseball statistics//EN"
  "http://metalab.unc.edu/xml/dtds/baseball.dtd">
```

You may have noticed that many HTML editors such as BBEdit automatically place the following string at the beginning of every HTML file they create:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML//EN">
```

Now you know what this string means! It says the document follows a non-standards-body-approved (-) DTD for HTML produced by the W3C in the English language.


**Note**

Technically the W3C is not a standards organization because its membership is limited to corporations that pay its fees rather than to official government-approved bodies. It only publishes *recommendations* instead of *standards*. In practice, the distinction is irrelevant.

## Internal and External DTD Subsets

Although most documents consist of easily defined pieces, not all documents use a common template. Many documents may need to use standard DTDs such as the HTML 4.0 DTD while adding custom elements for their own use. Other documents may use only standard elements, but need to reorder them. For instance, one HTML page may have a `BODY` that must contain exactly one `H1` header followed by a `DL` definition list while another may have a `BODY` that contains many different headers, paragraphs, and images in no particular order. If a particular document has a different structure than other pages on the site, it can be useful to define its structure in the document itself rather than in a separate DTD. This approach also makes the document easier to edit.

To this end, a document can use both an internal and an external DTD. The internal declarations go inside square brackets at the end of the `<!DOCTYPE>` tag. For example, suppose you want a page that includes baseball statistics but also has a header and a footer. Such a document might look like Listing 8-15. The baseball information is pulled from the file `baseball.dtd`, which forms the external DTD subset. The definition of the root element `DOCUMENT` as well as the `TITLE` and `SIGNATURE` elements come from the internal DTD subset included in the document itself. This is a little unusual. More commonly, the more generic pieces are likely to be part of an external DTD while the internal pieces are more topic-specific.

### Listing 8-15: A baseball document whose DTD has both an internal and an external subset

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT SYSTEM "baseball.dtd" [
  <!ELEMENT DOCUMENT (TITLE, SEASON, SIGNATURE)>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT COPYRIGHT (#PCDATA)>
  <!ELEMENT EMAIL (#PCDATA)>
  <!ELEMENT LAST_MODIFIED (#PCDATA)>
  <!ELEMENT SIGNATURE (COPYRIGHT, EMAIL, LAST_MODIFIED)>
]>

<DOCUMENT>
  <TITLE>1998 Major League Baseball Statistics</TITLE>
  <SEASON>
    <YEAR>1998</YEAR>
    <LEAGUE>
      <LEAGUE_NAME>National</LEAGUE_NAME>
      <DIVISION>
        <DIVISION_NAME>East</DIVISION_NAME>
      </DIVISION>
    </LEAGUE>
  </SEASON>
  <SIGNATURE>
    <COPYRIGHT>© 1998 Major League Baseball</COPYRIGHT>
    <EMAIL>mlb@mlb.com</EMAIL>
    <LAST_MODIFIED>2000-01-01</LAST_MODIFIED>
  </SIGNATURE>
</DOCUMENT>
```

*Continued*

## Listing 8-15 (continued)

```
<TEAM>
  <TEAM_CITY>Atlanta</TEAM_CITY>
  <TEAM_NAME>Braves</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>Florida</TEAM_CITY>
  <TEAM_NAME>Marlins</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>Montreal</TEAM_CITY>
  <TEAM_NAME>Expos</TEAM_NAME>
</TEAM>
<TEAM>
  <TEAM_CITY>New York</TEAM_CITY>
  <TEAM_NAME>Mets</TEAM_NAME>
</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>
    </TEAM>
</DIVISION>
<DIVISION>
    <DIVISION_NAME>West</DIVISION_NAME>
    <TEAM>
        <TEAM_CITY>Anaheim</TEAM_CITY>
        <TEAM_NAME>Angels</TEAM_NAME>
    </TEAM>
</DIVISION>
</LEAGUE>
</SEASON>
<SIGNATURE>
    <COPYRIGHT>Copyright 1999 Elliotte Rusty Harold</COPYRIGHT>
    <EMAIL>elharo@metalab.unc.edu</EMAIL>
    <LAST_MODIFIED>March 10, 1999</LAST_MODIFIED>
</SIGNATURE>
</DOCUMENT>

```

In the event of a conflict between elements of the same name in the internal and external DTD subsets, the elements declared internally take precedence. This precedence provides a crude, partial inheritance mechanism. For example, suppose you want to override the definition of a `PLAYER` element so that it can only contain batting statistics while disallowing pitching statistics. You could use most of the same declarations in the baseball DTD, changing the `PLAYER` element as follows:

```

<!DOCTYPE SEASON SYSTEM "baseball.dtd" [
    <!ELEMENT PLAYER (GIVEN_NAME, SURNAME, P, G,
        GS, AB?, R?, H?, D?, T?, HR?, RBI?, SB?, CS?,
        SH?, SF?, E?, BB?, S?, HBP?)
    >
]>

```

## Summary

In this chapter, you learned how to use a DTD to describe the structure of a document, that is, both the required and optional elements it contains and how those elements relate to one another. In particular you learned:

- ♦ A document type definition (DTD) provides a list of the elements, tags, attributes, and entities contained in the document, and their relationships to one another.
- ♦ A document's prolog may contain a document type declaration that specifies the root element and contains a DTD. This is placed between the XML declaration and before where the actual document begins. It is delimited by `<!DOCTYPE ROOT [ and ]>`, where `ROOT` is the name of the root element.

- ♦ DTDs lay out the permissible tags and the structure of a document. A document that adheres to the rules of its DTD is said to be valid.
- ♦ Element type declarations declare the name and children of an element.
- ♦ Children separated by commas in an element type declaration must appear in the same order in that element inside the document.
- ♦ A plus sign means one or more instances of the element may appear.
- ♦ An asterisk means zero or more instances of the element may appear.
- ♦ A question mark means zero or one instances of the child may appear.
- ♦ A vertical bar means one element or another is to be used.
- ♦ Parentheses group child elements to allow for more detailed element declarations.
- ♦ Mixed content contains both elements and parsed character data but limits the structure you can impose on the parent element.
- ♦ Empty elements are declared with the `EMPTY` keyword.
- ♦ Comments make DTDs much more legible.
- ♦ External DTDs can be located using the `SYSTEM` keyword and a URL in the document type declaration.
- ♦ Standard DTDs can be located using the `PUBLIC` keyword in the document type declaration.
- ♦ Declarations in the internal DTD subset override conflicting declarations in the external DTD subset

In the next chapter, you learn more about DTDs, including how entity references provide replacement text and how to separate DTDs from the documents they describe so they can be easily shared between documents. You also learn how to use multiple DTDs to describe a single document.

