EXHIBIT A

100%

ONE HUNDRED PERCENT

COMPREHENSIVE
AUTHORITATIVE
WHAT YOU NEED

ONE HUNDRED PERCENT

Master the simple yet powerful new markup language that's revolutionizing the Web

Build well-formed, sensibly organized Web documents

Create entirely new markup languages to fit your own needs

XML™ Bible

CD-ROM INSIDE!

- Code for every numbered listing in the book and additional examples
- XML browsers and tools
- Relevant W3C standards

Elliotte Rusty Harold

XML™ Bible

TASC, INC.
TECHNICAL LIBRARY
55 WALKERS BROOK DR
READING, MA 01867-3238

39804

QA 76.76 .H94 H34 1999

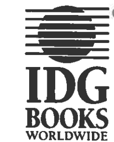
Harold, Elliotte Rusty.

XML bible

991378

XML™ Bible

Elliotte Rusty Harold



IDG Books Worldwide, Inc.
An International Data Group Company

Foster City, CA ✦ Chicago, IL ✦ Indianapolis, IN ✦ New York, NY

XML™ Bible

Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404
www.idgbooks.com (IDG Books Worldwide Web site)

Copyright © 1999 IDG Books Worldwide, Inc. All rights reserved. No part of this book, including interior design, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

ISBN: 0-7645-3236-7

Printed in the United States of America

10 9 8 7 6 5

1B/QU/QR/QQ/IN

Distributed in the United States by IDG Books Worldwide, Inc.

Distributed by CDG Books Canada Inc. for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by IDG Books Australia Publishing Corporation Pty. Ltd. for Australia and New Zealand; by TransQuest Publishers Pte Ltd. for Singapore, Malaysia, Thailand, Indonesia, and Hong Kong; by Gotop Information Inc. for Taiwan; by ICG Muse, Inc. for Japan; by Intersoft for South Africa; by Eyrolles for France; by International Thomson Publishing for Germany, Austria and Switzerland; by Distribuidora Cuspide for Argentina; by LR International for Brazil; by Galileo Libros for Chile; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Contemporanea de Ediciones for Venezuela; by Express Computer Distributors for the Caribbean and West Indies; by Micronesia Media Distributor, Inc. for Micronesia; by Chips Computadoras S.A. de C.V. for Mexico; by Editorial Norma de Panama S.A. for Panama; by American Bookshops for Finland.

For general information on IDG Books Worldwide's books in the U.S., please call our Consumer Customer Service department at 800-762-2974. For reseller information, including discounts and premium sales, please call our Reseller Customer Service department at 800-434-3422.

For information on where to purchase IDG Books Worldwide's books outside the U.S., please contact our International Sales department at 317-596-5530 or fax 317-572-4002.

For consumer information on foreign language translations, please contact our Customer Service department at 1-800-434-3422, fax 317-572-4002, or e-mail rights@idgbooks.com.

For information on licensing foreign or domestic rights, please phone +1-650-653-7098.

For sales inquiries and special prices for bulk quantities, please contact our Sales department at 800-762-2974 or write to the address above.

For information on using IDG Books Worldwide's books in the classroom or for ordering examination copies, please contact our Educational Sales department at 800-434-2086 or fax 317-572-4005.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 650-653-7000 or fax 650-653-7500.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

Library of Congress Cataloging-in-Publication Data

Harold, Elliote Rusty.
XML bible / Elliote Rusty Harold.
p. cm.
ISBN 0-7645-3236-7 (alk. paper)
1. XML (Document markup language) I. Title.
QA76.76.H94H34 1999 99-31021
005.7'2–dc21 CIP

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK. THE PUBLISHER AND AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THERE ARE NO WARRANTIES WHICH EXTEND BEYOND THE DESCRIPTIONS CONTAINED IN THIS PARAGRAPH. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ACCURACY AND COMPLETENESS OF THE INFORMATION PROVIDED HEREIN AND THE OPINIONS STATED HEREIN ARE NOT GUARANTEED OR WARRANTED TO PRODUCE ANY PARTICULAR RESULTS, AND THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY INDIVIDUAL. NEITHER THE PUBLISHER NOR AUTHOR SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

Trademarks: For Dummies, Dummies Man, A Reference for the Rest of Us!, The Dummies Way, Dummies Daily, and related trade dress are registered trademarks or trademarks of IDG Books Worldwide, Inc. in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. IDG Books Worldwide is not associated with any product or vendor mentioned in this book.

is a registered trademark under exclusive license to IDG Books Worldwide, Inc. from International Data Group, Inc.

CHAPTER 5

Attributes, Empty Tags, and XSL

In This Chapter

Attributes

Attributes versus elements

Empty tags

XSL

You can encode a given set of data in XML in nearly an infinite number of ways. There's no one right way to do it although some ways are more right than others, and some are more appropriate for particular uses. In this chapter, we explore a different solution to the problem of marking up baseball statistics in XML, carrying over the baseball example from the previous chapter. Specifically, we will address the use of attributes to store information and empty tags to define element positions. In addition, since CSS doesn't work well with content-less XML elements of this form, we'll examine an alternative — and more powerful — style sheet language called XSL.

Attributes

In the last chapter, all data was categorized into the name of a tag or the contents of an element. This is a straightforward and easy-to-understand approach, but it's not the only one. As in HTML, XML elements may have attributes. An attribute is a name-value pair associated with an element. The name and the value are each strings, and no element may contain two attributes with the same name.

You're already familiar with attribute syntax from HTML. For example, consider this <IMG> tag:

```
<IMG SRC=cup.gif WIDTH=89 HEIGHT=67 ALT="Cup
of coffee">
```

It has four attributes, the SRC attribute whose value is cup.gif, the WIDTH attribute whose value is 89, the HEIGHT attribute whose value is 67, and the ALT attribute whose value is Cup of coffee. However, in XML-unlike HTML-attribute values must always be quoted and start tags must have matching close tags. Thus, the XML equivalent of this tag is:

```
<IMG SRC="cup.gif" WIDTH="89" HEIGHT="67" ALT="Cup of coffee">
</IMG>
```

Note

Another difference between HTML and XML is that XML assigns no particular meaning to the IMG tag and its attributes. In particular, there's no guarantee that an XML browser will interpret this tag as an instruction to load and display the image in the file cup.gif.

You can apply attribute syntax to the baseball example quite easily. This has the advantage of making the markup somewhat more concise. For example, instead of containing a YEAR child element, the SEASON element only needs a YEAR attribute.

```
<SEASON YEAR="1998">
</SEASON>
```

On the other hand, LEAGUE should be a child of the SEASON element rather than an attribute. For one thing, there are two leagues in a season. Anytime there's likely to be more than one of something child elements are called for. Attribute names must be unique within an element. Thus you should not, for example, write a SEASON element like this:

```
<SEASON YEAR="1998" LEAGUE="National" League="American">
</SEASON>
```

The second reason LEAGUE is naturally a child element rather than an attribute is that it has substructure; it is subdivided into DIVISION elements. Attribute values are flat text. XML elements can conveniently encode structure-attribute values cannot.

However, the name of a league is unstructured, flat text; and there's only one name per league so LEAGUE elements can easily have a NAME attribute instead of a LEAGUE_NAME child element:

```
<LEAGUE NAME="National League">
</LEAGUE>
```

Since an attribute is more closely tied to its element than a child element is, you don't run into problems by using NAME instead of LEAGUE_NAME for the name of the attribute. Divisions and teams can also have NAME attributes without any fear of confusion with the name of a league. Since a tag can have more than one attribute (as long as the attributes have different names), you can make a team's city an attribute as well, as shown below:


```
<LEAGUE NAME="American League">
  <DIVISION NAME="East">
   <TEAM NAME="Orioles" CITY="Baltimore"></TEAM>
   <TEAM NAME="Red Sox" CITY="Boston"></TEAM>
   <TEAM NAME="Yankees" CITY="New York"></TEAM>
   <TEAM NAME="Devil Rays" CITY="Tampa Bay"></TEAM>
   <TEAM NAME="Blue Jays" CITY="Toronto"></TEAM>
  </DIVISION>
</LEAGUE>
```

Players will have a lot of attributes if you choose to make each statistic an attribute. For example, here are Joe Girardi's 1998 statistics as attributes:

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2">
</PLAYER>
```

Listing 5-1 uses this new attribute style for a complete XML document containing the baseball statistics for the 1998 major league season. It displays the same information (i.e., two leagues, six divisions, 30 teams, and nine players) as does Listing 4-1 in the last chapter. It is merely marked up differently. Figure 5-1 shows this document loaded into Internet Explorer 5.0 without a style sheet.

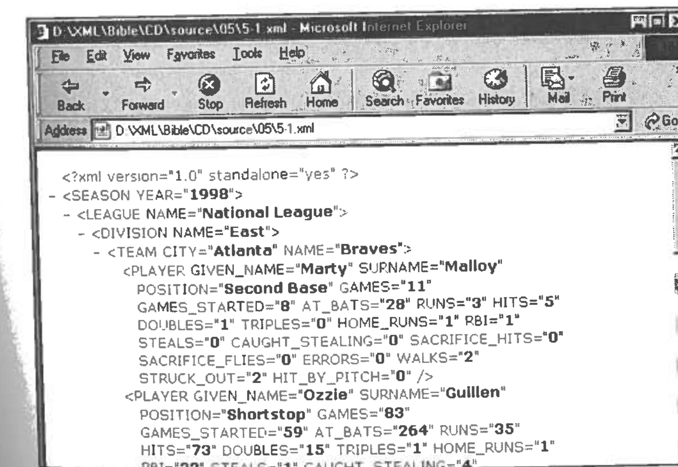


Figure 5-1: The 1998 major league baseball statistics using attributes for most information.

Listing 5-1: A complete XML document that uses attributes to store baseball statistics

```
<?xml version="1.0" standalone="yes"?>
<SEASON YEAR="1998">
  <LEAGUE NAME="National League">
    <DIVISION NAME="East">
      <TEAM CITY="Atlanta" NAME="Braves">
        <PLAYER GIVEN_NAME="Marty" SURNAME="Malloy"
         POSITION="Second Base" GAMES="11" GAMES_STARTED="8"
         AT_BATS="28" RUNS="3" HITS="5" DOUBLES="1"
         TRIPLES="0" HOME_RUNS="1" RBI="1" STEALS="0"
         CAUGHT_STEALING="0" SACRIFICE_HITS="0"
         SACRIFICE_FLIES="0" ERRORS="0" WALKS="2"
         STRUCK_OUT="2" HIT_BY_PITCH="0">
        </PLAYER>
        <PLAYER GIVEN_NAME="Ozzie" SURNAME="Guillen"
         POSITION="Shortstop" GAMES="83" GAMES_STARTED="59"
         AT_BATS="264" RUNS="35" HITS="73" DOUBLES="15"
         TRIPLES="1" HOME_RUNS="1" RBI="22" STEALS="1"
         CAUGHT_STEALING="4" SACRIFICE_HITS="4"
         SACRIFICE_FLIES="2" ERRORS="6" WALKS="24"
         STRUCK_OUT="25" HIT_BY_PITCH="1">
        </PLAYER>
        <PLAYER GIVEN_NAME="Danny" SURNAME="Bautista"
         POSITION="Outfield" GAMES="82" GAMES_STARTED="27"
         AT_BATS="144" RUNS="17" HITS="36" DOUBLES="11"
         TRIPLES="0" HOME_RUNS="3" RBI="17" STEALS="1"
         CAUGHT_STEALING="0" SACRIFICE_HITS="3"
         SACRIFICE_FLIES="2" ERRORS="2" WALKS="7"
         STRUCK_OUT="21" HIT_BY_PITCH="0">
        </PLAYER>
        <PLAYER GIVEN_NAME="Gerald" SURNAME="Williams"
         POSITION="Outfield" GAMES="129" GAMES_STARTED="51"
         AT_BATS="266" RUNS="46" HITS="81" DOUBLES="18"
         TRIPLES="3" HOME_RUNS="10" RBI="44" STEALS="11"
         CAUGHT_STEALING="5" SACRIFICE_HITS="2"
         SACRIFICE_FLIES="1" ERRORS="5" WALKS="17"
         STRUCK_OUT="48" HIT_BY_PITCH="3">
        </PLAYER>
        <PLAYER GIVEN_NAME="Tom" SURNAME="Glavine"
         POSITION="Starting Pitcher" GAMES="33"
         GAMES_STARTED="33" WINS="20" LOSSES="6" SAVES="0"
         COMPLETE_GAMES="4" SHUT_OUTS="3" ERA="2.47"
         INNINGS="229.1" HOME_RUNS_AGAINST="13"
         RUNS_AGAINST="67" EARNED_RUNS="63" HIT_BATTER="2"
         WILD_PITCHES="3" BALK="0" WALKED_BATTER="74"
         STRUCK_OUT_BATTER="157">
        </PLAYER>
        <PLAYER GIVEN_NAME="Javier" SURNAME="Lopez"
         POSITION="Catcher" GAMES="133" GAMES_STARTED="124"
         AT_BATS="489" RUNS="73" HITS="139" DOUBLES="21"
         TRIPLES="1" HOME_RUNS="34" RBI="106" STEALS="5"
```



```
         CAUGHT_STEALING="3" SACRIFICE_HITS="1"
         SACRIFICE_FLIES="8" ERRORS="5" WALKS="30"
         STRUCK_OUT="85" HIT_BY_PITCH="6">
        </PLAYER>
        <PLAYER GIVEN_NAME="Ryan" SURNAME="Klesko"
         POSITION="Outfield" GAMES="129" GAMES_STARTED="124"
         AT_BATS="427" RUNS="69" HITS="117" DOUBLES="29"
         TRIPLES="1" HOME_RUNS="18" RBI="70" STEALS="5"
         CAUGHT_STEALING="3" SACRIFICE_HITS="0"
         SACRIFICE_FLIES="4" ERRORS="2" WALKS="56"
         STRUCK_OUT="66" HIT_BY_PITCH="3">
        </PLAYER>
        <PLAYER GIVEN_NAME="Andres" SURNAME="Galarraga"
         POSITION="First Base" GAMES="153" GAMES_STARTED="151"
         AT_BATS="555" RUNS="103" HITS="169" DOUBLES="27"
         TRIPLES="1" HOME_RUNS="44" RBI="121" STEALS="7"
         CAUGHT_STEALING="6" SACRIFICE_HITS="0"
         SACRIFICE_FLIES="5" ERRORS="11" WALKS="63"
         STRUCK_OUT="146" HIT_BY_PITCH="25">
        </PLAYER>
        <PLAYER GIVEN_NAME="Wes" SURNAME="Helms"
         POSITION="Third Base" GAMES="7" GAMES_STARTED="2"
         AT_BATS="13" RUNS="2" HITS="4" DOUBLES="1"
         TRIPLES="0" HOME_RUNS="1" RBI="2" STEALS="0"
         CAUGHT_STEALING="0" SACRIFICE_HITS="0"
         SACRIFICE_FLIES="0" ERRORS="1" WALKS="0"
         STRUCK_OUT="4" HIT_BY_PITCH="0">
        </PLAYER>
      </TEAM>
      <TEAM CITY="Florida" NAME="Marlins">
      </TEAM>
      <TEAM CITY="Montreal" NAME="Expos">
      </TEAM>
      <TEAM CITY="New York" NAME="Mets">
      </TEAM>
      <TEAM CITY="Philadelphia" NAME="Phillies">
      </TEAM>
    </DIVISION>
    <DIVISION NAME="Central">
      <TEAM CITY="Chicago" NAME="Cubs">
      </TEAM>
      <TEAM CITY="Cincinnati" NAME="Reds">
      </TEAM>
      <TEAM CITY="Houston" NAME="Astros">
      </TEAM>
      <TEAM CITY="Milwaukee" NAME="Brewers">
      </TEAM>
      <TEAM CITY="Pittsburgh" NAME="Pirates">
      </TEAM>
      <TEAM CITY="St. Louis" NAME="Cardinals">
      </TEAM>
    </DIVISION>
```

Continued

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 9 of 55

Listing 5-1 *(continued)*

```
<DIVISION NAME="West">
  <TEAM CITY="Arizona" NAME="Diamondbacks">
  </TEAM>
  <TEAM CITY="Colorado" NAME="Rockies">
  </TEAM>
  <TEAM CITY="Los Angeles" NAME="Dodgers">
  </TEAM>
  <TEAM CITY="San Diego" NAME="Padres">
  </TEAM>
  <TEAM CITY="San Francisco" NAME="Giants">
  </TEAM>
 </DIVISION>
</LEAGUE>
<LEAGUE NAME="American League">
 <DIVISION NAME="East">
  <TEAM CITY="Baltimore" NAME="Orioles">
  </TEAM>
  <TEAM CITY="Boston" NAME="Red Sox">
  </TEAM>
  <TEAM CITY="New York" NAME="Yankees">
  </TEAM>
  <TEAM CITY="Tampa Bay" NAME="Devil Rays">
  </TEAM>
  <TEAM CITY="Toronto" NAME="Blue Jays">
  </TEAM>
 </DIVISION>
 <DIVISION NAME="Central">
  <TEAM CITY="Chicago" NAME="White Sox">
  </TEAM>
  <TEAM CITY="Kansas City" NAME="Royals">
  </TEAM>
  <TEAM CITY="Detroit" NAME="Tigers">
  </TEAM>
  <TEAM CITY="Cleveland" NAME="Indians">
  </TEAM>
  <TEAM CITY="Minnesota" NAME="Twins">
  </TEAM>
 </DIVISION>
 <DIVISION NAME="West">
  <TEAM CITY="Anaheim" NAME="Angels">
  </TEAM>
  <TEAM CITY="Oakland" NAME="Athletics">
  </TEAM>
  <TEAM CITY="Seattle" NAME="Mariners">
  </TEAM>
  <TEAM CITY="Texas" NAME="Rangers">
  </TEAM>
 </DIVISION>
</LEAGUE>
</SEASON>
```


Listing 5-1 uses only attributes for player information. Listing 4-1 used only element content. There are intermediate approaches as well. For example, you could make the player's name part of element content while leaving the rest of the statistics as attributes, like this:

```
<P>
  On Tuesday <PLAYER GAMES="78" AT_BATS="254" RUNS="31"
  HITS="70" DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRIKE_OUTS="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2">Joe Girardi</PLAYER> struck out twice
  and...
</P>
```

This would include Joe Girardi's name in the text of a page while still making his statistics available to readers who want to look deeper, as a hypertext footnote or tool tip. There's always more than one way to encode the same data. Which way you pick generally depends on the needs of your specific application.

Attributes versus Elements

There are no hard and fast rules about when to use child elements and when to use attributes. Generally, you'll use whichever suits your application. With experience, you'll gain a feel for when attributes are easier than child elements and vice versa. Until then, one good rule of thumb is that the data itself should be stored in elements. Information about the data (meta-data) should be stored in attributes. And when in doubt, put the information in the elements.

To differentiate between data and meta-data, ask yourself whether someone reading the document would want to see a particular piece of information. If the answer is yes, then the information probably belongs in a child element. If the answer is no, then the information probably belongs in an attribute. If all tags were stripped from the document along with all the attributes, the basic information should still be present. Attributes are good places to put ID numbers, URLs, references, and other information not directly or immediately relevant to the reader. However, there are many exceptions to the basic principal of storing meta-data as attributes. These include:

- ✦ Attributes can't hold structure well.
- ✦ Elements allow you to include meta-meta-data (information about the information about the information).
- ✦ Not everyone always agrees on what is and isn't meta-data.
- ✦ Elements are more extensible in the face of future changes.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 10 of 55

Structured Meta-data

One important principal to remember is that elements can have substructure and attributes can't. This makes elements far more flexible, and may convince you to encode meta-data as child elements. For example, suppose you're writing a paper and you want to include a source for a fact. It might look something like this:

```
<FACT SOURCE="The Biographical History of Baseball,
Donald Dewey and Nicholas Acocella (New York: Carroll &amp;
Graf Publishers, Inc. 1995) p. 169">
  Josh Gibson is the only person in the history of baseball to
  hit a pitch out of Yankee Stadium.
</FACT>
```

Clearly the information "The Biographical History of Baseball, Donald Dewey and Nicholas Acocella (New York: Carroll & Graf Publishers, Inc. 1995) p. 169" is meta-data. It is not the fact itself. Rather it is information about the fact. However, the SOURCE attribute contains a lot of implicit substructure. You might find it more useful to organize the information like this:

```
<SOURCE>
  <AUTHOR>Donald Dewey</AUTHOR>
  <AUTHOR>Nicholas Acocella</AUTHOR>
  <BOOK>
    <TITLE>The Biographical History of Baseball</TITLE>
    <PAGES>169</PAGES>
    <YEAR>1995</YEAR>
  </BOOK>
</SOURCE>
```

Furthermore, using elements instead of attributes makes it straightforward to include additional information like the authors' e-mail addresses, a URL where an electronic copy of the document can be found, the title or theme of the particular issue of the journal, and anything else that seems important.

Dates are another common example. One common piece of meta-data about scholarly articles is the date the article was first received. This is important for establishing priority of discovery and invention. It's easy to include a DATE attribute in an ARTICLE tag like this:

```
<ARTICLE DATE="06/28/1969">
  Polymerase Reactions in Organic Compounds
</ARTICLE>
```

However, the DATE attribute has substructure signified by the /. Getting that structure out of the attribute value, however, is much more difficult than reading child elements of a DATE element, as shown below:


```
<DATE>
  <YEAR>1969</YEAR>
  <MONTH>06</MONTH>
  <DAY>28</DAY>
</DATE>
```

For instance, with CSS or XSL, it's easy to format the day and month invisibly so that only the year appears. For example, using CSS:

```
YEAR {display: inline}
MONTH {display: none}
DAY {display: none}
```

If the DATE is stored as an attribute, however, there's no easy way to access only part of it. You must write a separate program in a programming language like ECMAScript or Java that can parse your date format. It's easier to use the standard XML tools and child elements.

Furthermore, the attribute syntax is ambiguous. What does the date "10/11/1999" signify? In particular, is it October 11th or November 10th? Readers from different countries will interpret this data differently. Even if your parser understands one format, there's no guarantee the people entering the data will enter it correctly. The XML, by contrast, is unambiguous.

Finally, using DATE children rather than attributes allows more than one date to be associated with an element. For instance, scholarly articles are often returned to the author for revisions. In these cases, it can also be important to note when the revised article was received. For example:

```
<ARTICLE>
  <TITLE>
    Maximum Projectile Velocity in an Augmented Railgun
  </TITLE>
  <AUTHOR>Elliotte Harold</AUTHOR>
  <AUTHOR>Bruce Bukiet</AUTHOR>
  <AUTHOR>William Peter</AUTHOR>
  <DATE>
    <YEAR>1992</YEAR>
    <MONTH>10</MONTH>
    <DAY>29</DAY>
  </DATE>
  <DATE>
    <YEAR>1993</YEAR>
    <MONTH>10</MONTH>
    <DAY>26</DAY>
  </DATE>
</ARTICLE>
```

As another example, consider the ALT attribute of an IMG tag in HTML. This is limited to a single string of text. However, given that a picture is worth a thousand words, you might well want to replace an IMG with marked up text. For instance, consider the pie chart shown in Figure 5-2.

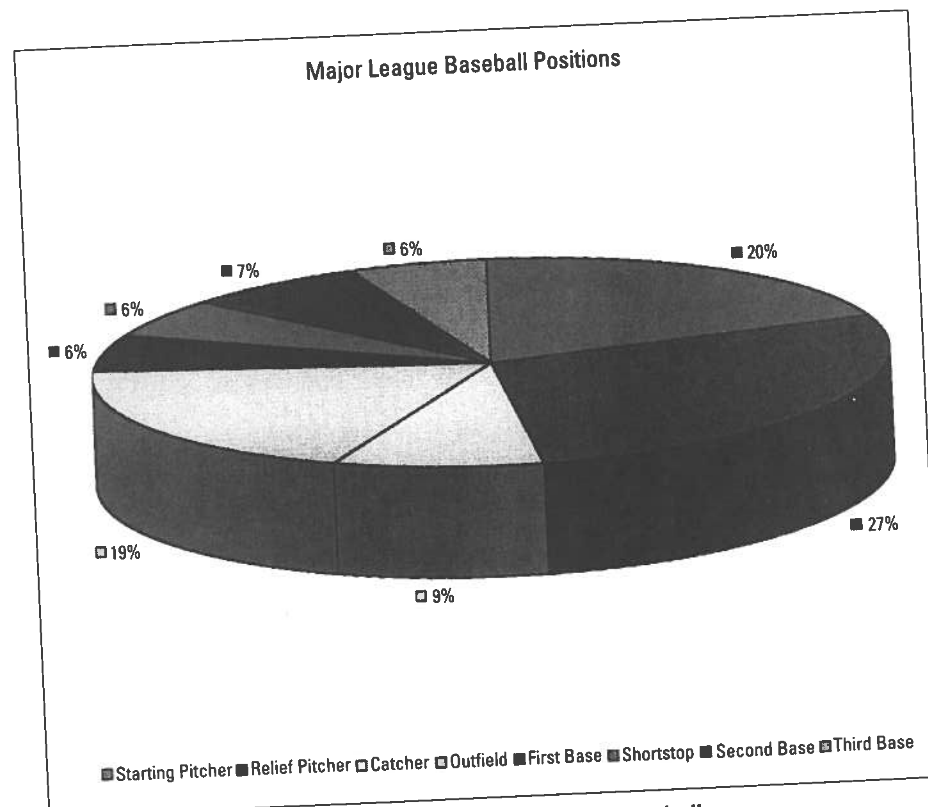


Figure 5-2: Distribution of positions in major league baseball

Using an ALT attribute, the best description of this picture you can provide is:

```
<IMG SRC="05021.gif"
     ALT="Pie Chart of Positions in Major League Baseball"
     WIDTH="819" HEIGHT="623">
</IMG>
```

However, with an ALT child element, you have more flexibility because you can embed markup. For example, you might provide a table of the relevant numbers instead of a pie chart.


```
<IMG SRC="05021.gif" WIDTH="819" HEIGHT="623">
  <ALT>
    <TABLE>
      <TR>
        <TD>Starting Pitcher</TD> <TD>242</TD> <TD>20%</TD>
      </TR>
      <TR>
        <TD>Relief Pitcher</TD> <TD>336</TD> <TD>27%</TD>
      </TR>
      <TR>
        <TD>Catcher</TD> <TD>104</TD> <TD>9%</TD>
      </TR>
      <TR>
        <TD>Outfield</TD> <TD>235</TD> <TD>19%</TD>
      </TR>
      <TR>
        <TD>First Base</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
      <TR>
        <TD>Shortstop</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
      <TR>
        <TD>Second Base</TD> <TD>88</TD> <TD>7%</TD>
      </TR>
      <TR>
        <TD>Third Base</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
    </TABLE>
  </ALT>
</IMG>
```

You might even provide the actual Postscript, SVG, or VML code to render the picture in the event that the bitmap image is not available.

Meta-Meta-Data

Using elements for meta-data also easily allows for meta-meta-data, or information about the information about the information. For example, the author of a poem may be considered to be meta-data about the poem. The language in which that author's name is written is data about the meta-data about the poem. This isn't a trivial concern, especially for distinctly non-Roman languages. For instance, is the author of the Odyssey Homer or Ωμηος? If you use elements, it's easy to write:

```
<POET LANGUAGE="English">Homer</POET>
<POET LANGUAGE="Greek">Ωμηος</POET>
```

However, if POET is an attribute rather than a child element, you're stuck with unwieldy constructs like this:

```
<POEM POET="Homer" POET_LANGUAGE="English"
 POEM_LANGUAGE="English">
  Tell me, O Muse, of the cunning man...
</POEM>
```

And it's even more bulky if you want to provide both the poet's English and Greek names.

```
<POEM POET_NAME_1="Homer" POET_LANGUAGE_1="English"
 POET_NAME_2="Ωμηος" POET_LANGUAGE_2="Greek"
 POEM_LANGUAGE="English">
  Tell me, O Muse, of the cunning man...
</POEM>
```

What's Your Meta-data Is Someone Else's Data

"Metaness" is in the mind of the beholder. Who is reading your document and why they are reading it determines what they consider to be data and what they consider to be meta-data. For example, if you're simply reading an article in a scholarly journal, then the author of the article is tangential to the information it contains. However, if you're sitting on a tenure and promotions committee scanning a journal to see who is publishing and who is not, then the names of the authors and the number of articles they've published may be more important to you than what they wrote (sad but true).

In fact, you may change your mind about what's meta and what's data. What's only tangentially relevant today, may become crucial to you next week. You can use style sheets to hide unimportant elements today, and change the style sheets to reveal them later. However, it's more difficult to later reveal information that was first stored in an attribute. Usually, this requires rewriting the document itself rather than simply changing the style sheet.

Elements Are More Extensible

Attributes are certainly convenient when you only need to convey one or two words of unstructured information. In these cases, there may genuinely be no current need for a child element. However, this doesn't preclude such a need in the future.

For instance, you may now only need to store the name of the author of an article, and you may not need to distinguish between the first and last names. However, in the future you may uncover a need to store first and last names, e-mail addresses, institution, snail mail address, URL, and more. If you've stored the author of the article as an element, then it's easy to add child elements to include this additional information.

Although any such change will probably require some revision of your documents, style sheets, and associated programs, it's still much easier to change a simple element to a tree of elements than it is to make an attribute a tree of elements. However, if you used an attribute, then you're stuck. It's quite difficult to extend your attribute syntax beyond the region it was originally designed for.

Good Times to Use Attributes

Having exhausted all the reasons why you should use elements instead of attributes, I feel compelled to point out that there are nonetheless some times when attributes make sense. First of all, as previously mentioned, attributes are fully appropriate for very simple data without substructure that the reader is unlikely to want to see. One example is the HEIGHT and WIDTH attributes of an IMG. Although the values of these attributes may change if the image changes, it's hard to imagine how the data in the attribute could be anything more than a very short string of text. HEIGHT and WIDTH are one-dimensional quantities (in more ways than one) so they work well as attributes.

Furthermore, attributes are appropriate for simple information about the document that has nothing to do with the content of the document. For example, it is often useful to assign an ID attribute to each element. This is a unique string possessed only by one element in the document. You can then use this string for a variety of tasks including linking to particular elements of the document, even if the elements move around as the document changes over time. For example:

```
<SOURCE ID="S1">
  <AUTHOR ID="A1">Donald Dewey</AUTHOR>
  <AUTHOR ID="A2">Nicholas Acocella</AUTHOR>
  <BOOK ID="B1">
    <TITLE ID="B2">
      The Biographical History of Baseball
    </TITLE>
    <PAGES ID="B3">169</PAGES>
    <YEAR ID="B4">1995</YEAR>
  </BOOK>
</SOURCE>
```

ID attributes make links to particular elements in the document possible. In this way, they can serve the same purpose as the NAME attribute of HTML's A elements. Other data associated with linking—HREFs to link to, SRCs to pull images and binary data from, and so forth—also work well as attributes.

You'll see more examples of this when XLL, the Extensible Linking Language, is discussed in Chapter 16, *XLinks*, and Chapter 17, *XPointers*.

Attributes are also often used to store document-specific style information. For example, if TITLE elements are generally rendered as bold text but if you want to make just one TITLE element both bold and italic, you might write something like this:

```
<TITLE style="font-style: italic">Significant Others</TITLE>
```

This enables the style information to be embedded without changing the tree structure of the document. While ideally you'd like to use a separate element, this scheme gives document authors somewhat more control when they cannot add elements to the tag set they're working with. For example, the Webmaster of a site might require the use of a particular DTD and not want to allow everyone to modify the DTD. Nonetheless, they want to allow them to make minor adjustments to individual pages. Use this scheme with restraint, however, or you'll soon find yourself back in the HTML hell XML was supposed to save us from, where formatting is freely intermixed with meaning and documents are no longer maintainable.

The final reason to use attributes is to maintain compatibility with HTML. To the extent that you're using tags that at least look similar to HTML such as <IMG>, <P>, and <TD>, you might as well employ the standard HTML attributes for these tags. This has the double advantage of enabling legacy browsers to at least partially parse and display your document, and of being more familiar to the people writing the documents.

Empty Tags

Last chapter's no-attribute approach was an extreme position. It's also possible to swing to the other extreme— storing all the information in the attributes and none in the content. In general, I don't recommend this approach. Storing all the information in element content—while equally extreme—is much easier to work with in practice. However, this section entertains the possibility of using only attributes for the sake of elucidation.

As long as you know the element will have no content, you can use empty tags as a short cut. Rather than including both a start and an end tag you can include one empty tag. Empty tags are distinguished from start tags by a closing /> instead of simply a closing >. For instance, instead of <PLAYER></PLAYER> you would write <PLAYER/>.

Empty tags may contain attributes. For example, here's an empty tag for Joe Girardi with several attributes:

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2"/>
```


XML parsers treat this identically to the non-empty equivalent. This PLAYER element is precisely equal (though not identical) to the previous PLAYER element formed with an empty tag.

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2"></PLAYER>
```

The difference between <PLAYER/> and <PLAYER></PLAYER> is syntactic sugar, and nothing more. If you don't like the empty tag syntax, or find it hard to read, you don't have to use it.

XSL

Attributes are visible in an XML source view of the document as shown in Figure 5-1. However, once a CSS style sheet is applied the attributes disappear. Figure 5-3 shows Listing 5-1 once the baseball stats style sheet from the previous chapter is applied. It looks like a blank document because CSS styles only apply to element content, not to attributes. If you use CSS, any data you want to display to the reader should be part of an element's content rather than one of its attributes.

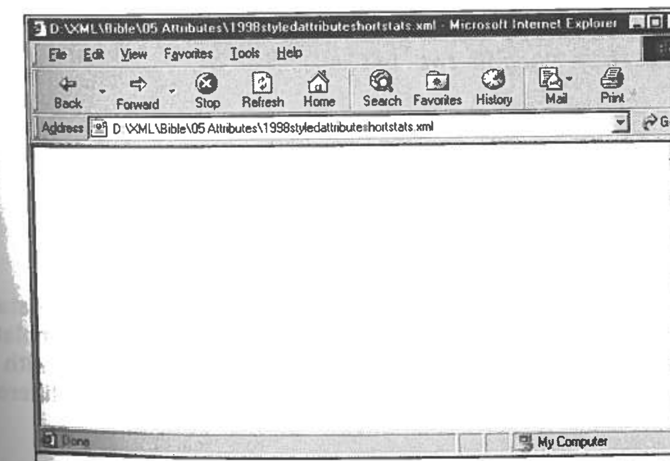


Figure 5-3: A blank document is displayed when CSS is applied to an XML document whose elements do not contain any character data.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 14 of 55

However, there is an alternative style sheet language that does allow you to access and display attribute data. This language is the Extensible Style Language (XSL); and it is also supported by Internet Explorer 5.0, at least in part. XSL is divided into two sections, transformations and formatting.

The transformation part of XSL enables you to replace one tag with another. You can define rules that replace your XML tags with standard HTML tags, or with HTML tags plus CSS attributes. You can also do a lot more including reordering the elements in the document and adding additional content that was never present in the XML document.

The formatting part of XSL defines an extremely powerful view of documents as pages. XSL formatting enables you to specify the appearance and layout of a page including multiple columns, text flow around objects, line spacing, assorted font properties, and more. It's designed to be powerful enough to handle automated layout tasks for both the Web and print from the same source document. For instance, XSL formatting would allow one XML document containing show times and advertisements to generate both the print and online editions of a local newspaper's television listings. However, IE 5.0 and most other tools do not yet support XSL formatting. Therefore, in this section I'll focus on XSL transformations.

XSL formatting is discussed in Chapter 15, *XSL Formatting Objects*.

XSL Style Sheet Templates

An XSL style sheet contains templates into which data from the XML document is poured. For example, one template might look something like this:

```
<HTML>
  <HEAD>
    <TITLE>
      XSL Instructions to get the title
    </TITLE>
  </HEAD>
  <H1>XSL Instructions to get the title</H1>
  <BODY>
    XSL Instructions to get the statistics
  </BODY>
</HTML>
```

The italicized sections will be replaced by particular XSL elements that copy data from the underlying XML document into this template. You can apply this template to many different data sets. For instance, if the template is designed to work with the baseball example, then the same style sheet can display statistics from different seasons.

This may remind you of some server-side include schemes for HTML. In fact, this is very much like server-side includes. However, the actual transformation of the source XML document and XSL style sheet takes place on the client rather than on the server. Furthermore, the output document does not have to be HTML. It can be any well-formed XML.

XSL instructions can retrieve any data stored in the elements of the XML document. This includes element content, element names, and, most importantly for our example, element attributes. Particular elements are chosen by a pattern that considers the element's name, its value, its attributes' names and values, its absolute and relative position in the tree structure of the XML document, and more. Once the data is extracted from an element, it can be moved, copied, and manipulated in a variety of ways. We won't cover everything you can do with XML transformations in this brief introduction. However, you will learn to use XSL to write some pretty amazing documents that can be viewed on the Web right away.

Chapter 14, *XSL Transformations*, covers XSL transformations in depth.

The Body of the Document

Let's begin by looking at a simple example and applying it to the XML document with baseball statistics shown in Listing 5-1. Listing 5-2 is an XSL style sheet. This style sheet provides the HTML mold into which XML data will be poured.

Listing 5-2: An XSL style sheet

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>
        <H1>Major League Baseball Statistics</H1>

        <HR></HR>
        Copyright 1999
        <A HREF="http://www.macfaq.com/personal.html">
         Elliotte Rusty Harold
        </A>
```

Continued

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 15 of 55

ng 5-2 (continued)

```
    <BR />
    <A HREF="mailto:elharo@metalab.unc.edu">
     elharo@metalab.unc.edu
    </A>

  </BODY>
 </HTML>
/xsl:template>

sl:stylesheet>
```

embles an HTML file included inside an `xsl:template` element. In other
s its structure looks like this:

```
xml version="1.0"?>
sl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

 <xsl:template match="/">
   HTML file goes here
 </xsl:template>

/xsl:stylesheet>
```

ing 5-2 is not only an XSL style sheet; it's also a well-formed XML document.
egins with an XML declaration. The root element of this document is `xsl:`
lesheet. This style sheet contains a single template for the XML data
oded as an `xsl:template` element. The `xsl:template` element has a `match`
ribute with the value / and its content is a well-formed HTML document. It's
t a coincidence that the output HTML is well-formed. Because the HTML must
st be part of an XSL style sheet, and because XSL style sheets are well-formed
L documents, all the HTML in a XSL style sheet must be well-formed.

e Web browser tries to match parts of the XML document against each
l:template element. The / template matches the root of the document; that is
e entire document itself. The browser reads the template and inserts data from
e XML document where indicated by XSL instructions. However, this particular
mplate contains no XSL instructions, so its contents are merely copied verbatim
to the Web browser, producing the output you see in Figure 5-4. Notice that Figure
-4 does not display any data from the XML document, only from the XSL template.

Attaching the XSL style sheet of Listing 5-2 to the XML document in Listing 5-1 is straightforward. Simply add a `<?xml-stylesheet?>` processing instruction with a `type` attribute with value `text/xsl` and an `href` attribute that points to the style sheet between the XML declaration and the root element. For example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="5-2.xsl"?>
<SEASON YEAR="1998">
...
```

This is the same way a CSS style sheet is attached to a document. The only difference is that the `type` attribute is `text/xsl` instead of `text/css`.

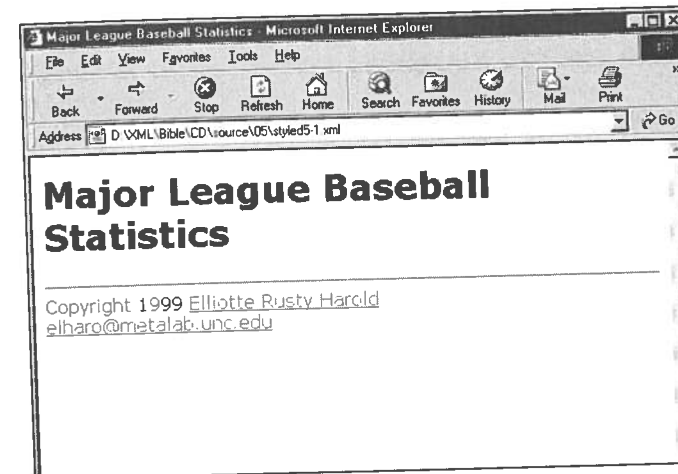


Figure 5-4: The data from the XML document, not the XSL template, is missing after application of the XSL style sheet in Listing 5-2.

The Title

Of course there was something rather obvious missing from Figure 5-4 — the data! Although the style sheet in Listing 5-2 displays something (unlike the CSS style sheet of Figure 5-3) it doesn't show any data from the XML document. To add this, you need to use XSL instruction elements to copy data from the source XML document into the XSL template. Listing 5-3 adds the necessary XSL instructions to extract the `YEAR` attribute from the `SEASON` element and insert it in the `TITLE` and `H1` header of the resulting document. Figure 5-5 shows the rendered document.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 16 of 55

Listing 5-2 *(continued)*

```
        <BR />
        <A HREF="mailto:elharo@metalab.unc.edu">
         elharo@metalab.unc.edu
        </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

It resembles an HTML file included inside an `xsl:template` element. In other words its structure looks like this:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    HTML file goes here
  </xsl:template>

</xsl:stylesheet>
```

Listing 5-2 is not only an XSL style sheet; it's also a well-formed XML document. It begins with an XML declaration. The root element of this document is `xsl:stylesheet`. This style sheet contains a single template for the XML data encoded as an `xsl:template` element. The `xsl:template` element has a `match` attribute with the value `/` and its content is a well-formed HTML document. It's not a coincidence that the output HTML is well-formed. Because the HTML must first be part of an XSL style sheet, and because XSL style sheets are well-formed XML documents, all the HTML in a XSL style sheet must be well-formed.

The Web browser tries to match parts of the XML document against each `xsl:template` element. The `/` template matches the root of the document; that is the entire document itself. The browser reads the template and inserts data from the XML document where indicated by XSL instructions. However, this particular template contains no XSL instructions, so its contents are merely copied verbatim into the Web browser, producing the output you see in Figure 5-4. Notice that Figure 5-4 does not display any data from the XML document, only from the XSL template.

Attaching the XSL style sheet of Listing 5-2 to the XML document in Listing 5-1 is straightforward. Simply add a `<?xml-stylesheet?>` processing instruction with a `type` attribute with value `text/xsl` and an `href` attribute that points to the style sheet between the XML declaration and the root element. For example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="5-2.xsl"?>
<SEASON YEAR="1998">
...
```

This is the same way a CSS style sheet is attached to a document. The only difference is that the `type` attribute is `text/xsl` instead of `text/css`.

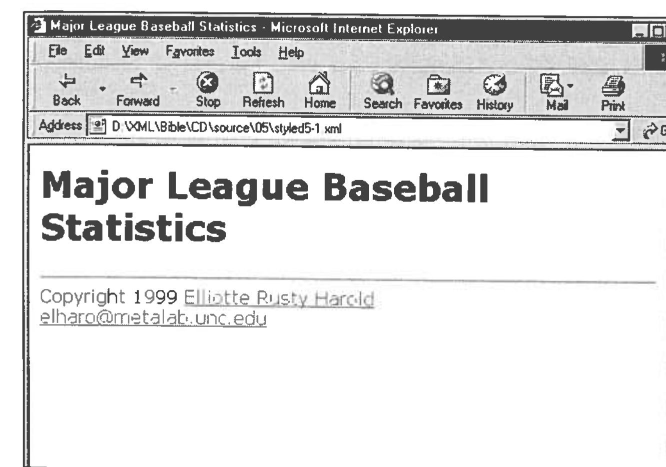


Figure 5-4: The data from the XML document, not the XSL template, is missing after application of the XSL style sheet in Listing 5-2.

The Title

Of course there was something rather obvious missing from Figure 5-4 — the data! Although the style sheet in Listing 5-2 displays something (unlike the CSS style sheet of Figure 5-3) it doesn't show any data from the XML document. To add this, you need to use XSL instruction elements to copy data from the source XML document into the XSL template. Listing 5-3 adds the necessary XSL instructions to extract the `YEAR` attribute from the `SEASON` element and insert it in the `TITLE` and `H1` header of the resulting document. Figure 5-5 shows the rendered document.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 17 of 55

Listing 5-3: An XSL style sheet with instructions to extract the SEASON element and YEAR attribute

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>
          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>
      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

The new XSL instructions that extract the YEAR attribute from the SEASON element are:

```
<xsl:for-each select="SEASON">
  <xsl:value-of select="@YEAR"/>
</xsl:for-each>
```

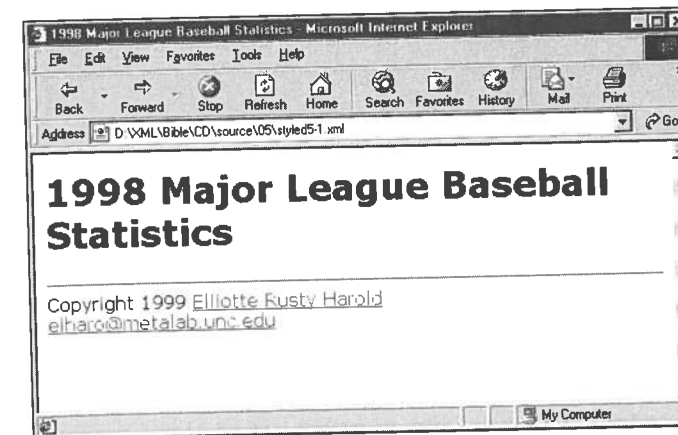



Figure 5-5: Listing 5-1 after application of the XSL style sheet in Listing 5-3

These instructions appear twice because we want the year to appear twice in the output document-once in the H1 header and once in the TITLE. Each time they appear, these instructions do the same thing. `<xsl:for-each select="SEASON">` finds all SEASON elements. `<xsl:value-of select="@YEAR"/>` inserts the value of the YEAR attribute of the SEASON element — that is, the string "1998" — found by `<xsl:for-each select="SEASON">`.

This is important, so let me say it again: xsl:for-each selects a particular XML element in the source document (Listing 5-1 in this case) from which data will be read. xsl:value-of copies a particular part of the element into the output document. You need both XSL instructions. Neither alone is sufficient.

XSL instructions are distinguished from output elements like HTML and H1 because the instructions are in the xsl namespace. That is, the names of all XSL elements begin with xsl:. The namespace is identified by the xmlns:xsl attribute of the root element of the style sheet. In Listings 5-2, 5-3, and all other examples in this book, the value of that attribute is http://www.w3.org/TR/WD-xsl.

Namespaces are covered in depth in Chapter 18, *Namespaces*.

Leagues, Divisions, and Teams

Next, let's add some XSL instructions to pull out the two LEAGUE elements. We'll map these to H2 headers. Listing 5-4 demonstrates. Figure 5-6 shows the document rendered with this style sheet.

Listing 5-4: An XSL style sheet with instructions to extract LEAGUE elements

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>
        </xsl:for-each>

      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

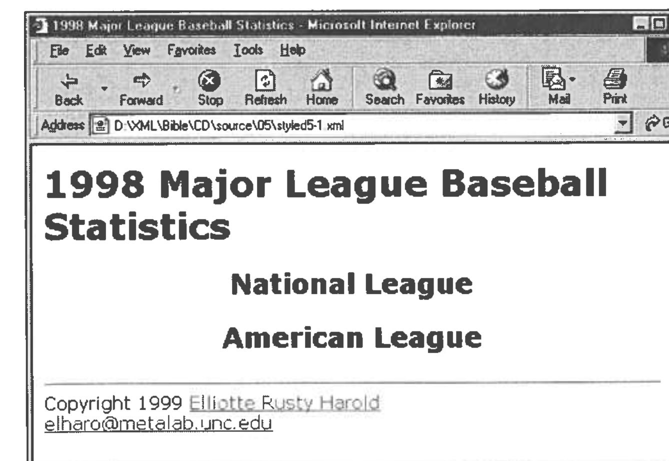



Figure 5-6: The league names are displayed as H2 headers when the XSL style sheet in Listing 5-4 is applied.

The key new materials are the nested `xsl:for-each` instructions

```
<xsl:for-each select="SEASON">
  <H1>
    <xsl:value-of select="@YEAR"/>
    Major League Baseball Statistics
  </H1>

  <xsl:for-each select="LEAGUE">
    <H2 ALIGN="CENTER">
      <xsl:value-of select="@NAME"/>
    </H2>
  </xsl:for-each>

</xsl:for-each>
```

The outermost instruction says to select the `SEASON` element. With that element selected, we then find the `YEAR` attribute of that element and place it between `<H1>` and `</H1>` along with the extra text `Major League Baseball Statistics`. Next, the browser loops through each `LEAGUE` child of the selected `SEASON` and places the value of its `NAME` attribute between `<H2 ALIGN="CENTER">` and `</H2>`. Although there's only one `xsl:for-each` matching a `LEAGUE` element, it loops over all the `LEAGUE` elements that are immediate children of the `SEASON` element. Thus, this template works for anywhere from zero to an indefinite number of leagues.

The same technique can be used to assign `H3` headers to divisions and `H4` headers to teams. Listing 5-5 demonstrates the procedure and Figure 5-7 shows the document rendered with this style sheet. The names of the divisions and teams are read from the XML data.

Listing 5-5: An XSL style sheet with instructions to extract DIVISION and TEAM elements

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>

          <xsl:for-each select="DIVISION">
            <H3 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
            </H3>

            <xsl:for-each select="TEAM">
              <H4 ALIGN="CENTER">
              <xsl:value-of select="@CITY"/>
              <xsl:value-of select="@NAME"/>
              </H4>
            </xsl:for-each>
          </xsl:for-each>

        </xsl:for-each>
      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
```



```
      Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
      elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

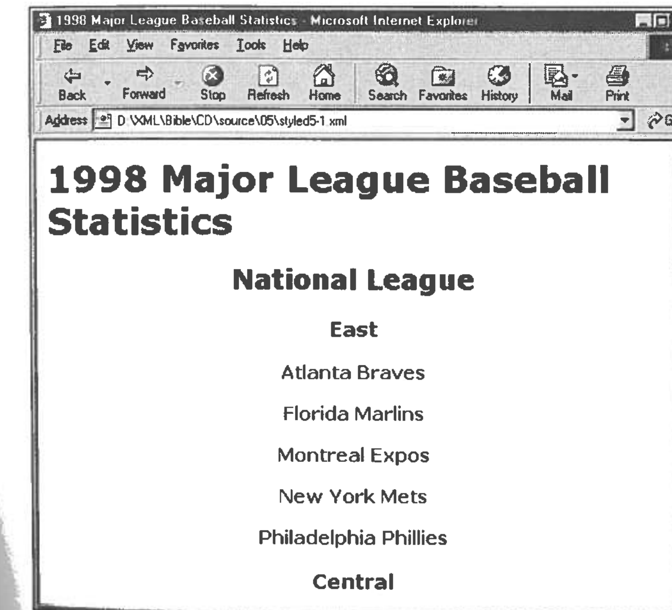


Figure 5-7: Divisions and team names are displayed after application of the XSL style sheet in Listing 5-5.

In the case of the TEAM elements, the values of both its CITY and NAME attributes are used as contents for the H4 header. Also notice that the nesting of the xsl:for-each elements that selects seasons, leagues, divisions, and teams mirrors the hierarchy of the document itself. That's not a coincidence. While other schemes are possible that don't require matching hierarchies, this is the simplest, especially for highly structured data like the baseball statistics of Listing 5-1.

Players

The next step is to add statistics for individual players on a team. The most natural way to do this is in a table. Listing 5-6 shows an XSL style sheet that arranges the players and their stats in a table. No new XSL elements are introduced. The same xsl:for-each and xsl:value-of elements are used on the PLAYER element and its attributes. The output is standard HTML table tags. Figure 5-8 displays the results.

Listing 5-6: An XSL style sheet that places players and their statistics in a table

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>

          <xsl:for-each select="DIVISION">
            <H3 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
            </H3>

            <xsl:for-each select="TEAM">
              <H4 ALIGN="CENTER">
              <xsl:value-of select="@CITY"/>
              <xsl:value-of select="@NAME"/>
              </H4>

              <TABLE>
```



```
              <THEAD>
               <TR>
                <TH>Player</TH><TH>P</TH><TH>G</TH>
                <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
                <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
                <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
                <TH>E</TH><TH>BB</TH><TH>SO</TH><TH>HBP</TH>
               </TR>
              </THEAD>
              <TBODY>
               <xsl:for-each select="PLAYER">
                <TR>
                 <TD>
                  <xsl:value-of select="@GIVEN_NAME"/>
                  <xsl:value-of select="@SURNAME"/>
                 </TD>
                 <TD><xsl:value-of select="@POSITION"/></TD>
                 <TD><xsl:value-of select="@GAMES"/></TD>
                 <TD>
                   <xsl:value-of select="@GAMES_STARTED"/>
                 </TD><xsl:value-of select="@AT_BATS"/></TD>
                 <TD><xsl:value-of select="@RUNS"/></TD>
                 <TD><xsl:value-of select="@HITS"/></TD>
                 <TD><xsl:value-of select="@DOUBLES"/></TD>
                 <TD><xsl:value-of select="@TRIPLES"/></TD>
                 <TD><xsl:value-of select="@HOME_RUNS"/></TD>
                 <TD><xsl:value-of select="@RBI"/></TD>
                 <TD><xsl:value-of select="@STEALS"/></TD>
                 <TD>
                  <xsl:value-of select="@CAUGHT_STEALING"/>
                 </TD>
                 <TD>
                  <xsl:value-of select="@SACRIFICE_HITS"/>
                 </TD>
                 <TD>
                  <xsl:value-of select="@SACRIFICE_FLIES"/>
                 </TD><xsl:value-of select="@ERRORS"/></TD>
                 <TD><xsl:value-of select="@WALKS"/></TD>
                 <TD>
                  <xsl:value-of select="@STRUCK_OUT"/>
                 </TD>
                 <TD>
                  <xsl:value-of select="@HIT_BY_PITCH"/>
                 </TD>
                </TR>
               </xsl:for-each>
              </TBODY>
             </TABLE>

            </xsl:for-each>
```

Continued

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 21 of 55

Listing 5-6 *(continued)*

```
          </xsl:for-each>

        </xsl:for-each>
      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

Separation of Pitchers and Batters

One discrepancy you might notice in Figure 5-8 is that the pitchers aren't handled properly. Throughout this chapter and Chapter 4, we've always given the pitchers a completely different set of statistics, whether those stats were stored in element content or attributes. Therefore, the pitchers really need a table that is separate from the other players. Before putting a player into the table, you must test whether he is or is not a pitcher. If his `POSITION` attribute contains the string "pitcher" then omit him. Then reverse the procedure in a second table that only includes pitchers-`PLAYER` elements whose `POSITION` attribute contains the string "pitcher".

To do this, you have to add additional code to the `xsl:for-each` element that selects the players. You don't select all players. Instead, you select those players whose `POSITION` attribute is not pitcher. The syntax looks like this:

```
<xsl:for-each select="PLAYER[(@POSITION != 'Pitcher')">
```

But because the XML document distinguishes between starting and relief pitchers, the true answer must test both cases:

```
<xsl:for-each select="PLAYER[(@POSITION != 'Starting Pitcher')
  $and$ (@POSITION != 'Relief Pitcher')]">
```


1998 Major League Baseball Statistics - Microsoft Internet Explorer

1998 Major League Baseball Statistics

National League

East

Atlanta Braves

Player	P	G	GS	AB	R	H	D	T	HR	RBI	S	CS	SH	SF	E	BB	SO	HBP
Marty Malloy	Second Base	11	8	28	3	5	1	0	1	1	0	0	0	0	0	2	2	0
Ozzie Guillen	Shortstop	83	59	264	35	73	15	1	1	22	1	4	4	2	6	24	25	1
Danny Bautista	Outfield	82	27	144	17	36	11	0	3	17	1	0	3	2	2	7	21	0
Gerald Williams	Outfield	129	51	266	46	81	18	3	10	44	11	5	2	1	5	17	48	3
Tom Glavine	Starting Pitcher	33	33															
Javier Lopez	Catcher	133	124	489	73	139	21	1	34	106	5	3	1	8	5	30	85	6
Ryan Klesko	Outfield	129	124	427	69	117	29	1	18	70	5	3	0	4	2	56	66	3
Andres	First Base	153	151	555	103	169	27	1	44	121	7	6	0	5	11	63	146	25

Figure 5-8: Player statistics are displayed after applying the XSL style sheet in Listing 5-6.

For the table of pitchers, you logically reverse this to the position being equal to either "Starting Pitcher" or "Relief Pitcher". (It is not sufficient to just change *not equal* to *equal*. You also have to change *and* to *or*.) The syntax looks like this:

```
<xsl:for-each select="PLAYER[(@POSITION = 'Starting Pitcher')
 $or$ (@POSITION = 'Relief Pitcher')]">
```

Note Only a single equals sign is used to test for equality rather than the double equals sign used in C and Java. That's because there's no equivalent of an assignment operator in XSL.

Listing 5-7 shows an XSL style sheet separating the batters and pitchers into two different tables. The pitchers' table adds columns for all the usual pitcher statistics. Listing 5-1 encodes in attributes: wins, losses, saves, shutouts, etc. Abbreviations are used for the column labels to keep the table to a manageable width. Figure 5-9 shows the results.

Listing 5-7: An XSL style sheet that separates batters and pitchers

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

        <xsl:for-each select="SEASON">
          <H1>
            <xsl:value-of select="@YEAR"/>
            Major League Baseball Statistics
          </H1>

          <xsl:for-each select="LEAGUE">
            <H2 ALIGN="CENTER">
              <xsl:value-of select="@NAME"/>
            </H2>

            <xsl:for-each select="DIVISION">
              <H3 ALIGN="CENTER">
              <xsl:value-of select="@NAME"/>
              </H3>

              <xsl:for-each select="TEAM">
                <H4 ALIGN="CENTER">
                <xsl:value-of select="@CITY"/>
                <xsl:value-of select="@NAME"/>
                </H4>

                <TABLE>
                 <CAPTION><B>Batters</B></CAPTION>
                 <THEAD>
                  <TR>
                   <TH>Player</TH><TH>P</TH><TH>G</TH>
                   <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
                   <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
                   <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
                   <TH>E</TH><TH>BB</TH><TH>SO</TH>
                   <TH>HBP</TH>
                  </TR>
                 </THEAD>
```



```
<TBODY>
 <xsl:for-each select="PLAYER[(@POSITION
   != 'Starting Pitcher')
  $and$ (@POSITION != 'Relief Pitcher')]">
  <TR>
   <TD>
    <xsl:value-of select="@GIVEN_NAME"/>
    <xsl:value-of select="@SURNAME"/>
   </TD>
   <TD><xsl:value-of select="@POSITION"/></TD>
   <TD><xsl:value-of select="@GAMES"/></TD>
   <TD>
     <xsl:value-of select="@GAMES_STARTED"/>
   </TD>
   <TD><xsl:value-of select="@AT_BATS"/></TD>
   <TD><xsl:value-of select="@RUNS"/></TD>
   <TD><xsl:value-of select="@HITS"/></TD>
   <TD><xsl:value-of select="@DOUBLES"/></TD>
   <TD><xsl:value-of select="@TRIPLES"/></TD>
   <TD>
     <xsl:value-of select="@HOME_RUNS"/>
   </TD>
   <TD><xsl:value-of select="@RBI"/></TD>
   <TD><xsl:value-of select="@STEALS"/></TD>
   <TD>
    <xsl:value-of select="@CAUGHT_STEALING"/>
   </TD>
   <TD>
    <xsl:value-of select="@SACRIFICE_HITS"/>
   </TD>
   <TD>
    <xsl:value-of select="@SACRIFICE_FLIES"/>
   </TD>
   <TD><xsl:value-of select="@ERRORS"/></TD>
   <TD><xsl:value-of select="@WALKS"/></TD>
   <TD>
    <xsl:value-of select="@STRUCK_OUT"/>
   </TD>
   <TD>
    <xsl:value-of select="@HIT_BY_PITCH"/>
   </TD>
  </TR>
 </xsl:for-each>  <!-- PLAYER -->
</TBODY>
</TABLE>

<TABLE>
  <CAPTION><B>Pitchers</B></CAPTION>
  <THEAD>
   <TR>
    <TH>Player</TH><TH>P</TH><TH>G</TH>
    <TH>GS</TH><TH>W</TH><TH>L</TH><TH>S</TH>
```

Continued

Listing 5-7 *(continued)*

```
<TH>CG</TH><TH>SO</TH><TH>ERA</TH>
<TH>IP</TH><TH>HR</TH><TH>R</TH><TH>ER</TH>
<TH>HB</TH><TH>WP</TH><TH>B</TH><TH>BB</TH>
<TH>K</TH>
</TR>
</THEAD>
<TBODY>
<xsl:for-each select="PLAYER[(@POSITION
 = 'Starting Pitcher')
$or$ (@POSITION = 'Relief Pitcher')]">
<TR>
 <TD>
  <xsl:value-of select="@GIVEN_NAME"/>
  <xsl:value-of select="@SURNAME"/>
 </TD>
 <TD><xsl:value-of select="@POSITION"/></TD>
 <TD><xsl:value-of select="@GAMES"/></TD>
 <TD>
  <xsl:value-of select="@GAMES_STARTED"/>
 </TD>
 <TD><xsl:value-of select="@WINS"/></TD>
 <TD><xsl:value-of select="@LOSSES"/></TD>
 <TD><xsl:value-of select="@SAVES"/></TD>
 <TD>
  <xsl:value-of select="@COMPLETE_GAMES"/>
 </TD>
 <TD>
  <xsl:value-of select="@SHUT_OUTS"/>
 </TD>
 <TD><xsl:value-of select="@ERA"/></TD>
 <TD><xsl:value-of select="@INNINGS"/></TD>
 <TD>
  <xsl:value-of select="@HOME_RUNS_AGAINST"/>
 </TD>
 <TD>
  <xsl:value-of select="@RUNS_AGAINST"/>
 </TD>
 <TD>
  <xsl:value-of select="@EARNED_RUNS"/>
 </TD>
 <TD>
  <xsl:value-of select="@HIT_BATTER"/>
 </TD>
 <TD>
  <xsl:value-of select="@WILD_PITCH"/>
 </TD>
 <TD><xsl:value-of select="@BALK"/></TD>
 <TD>
  <xsl:value-of select="@WALKED_BATTER"/>
 </TD>
 <TD>
```



```
              <xsl:value-of select="@STRUCK_OUT_BATTER"/>
             </TD>
            </TR>
           </xsl:for-each>  <!-- PLAYER -->
          </TBODY>
         </TABLE>

        </xsl:for-each> <!-- TEAM -->
       </xsl:for-each> <!-- DIVISION -->
     </xsl:for-each> <!-- LEAGUE -->
    </xsl:for-each> <!-- SEASON -->


     <HR></HR>
     Copyright 1999
     <A HREF="http://www.macfaq.com/personal.html">
      Elliotte Rusty Harold
     </A>
     <BR />
     <A HREF="mailto:elharo@metalab.unc.edu">
      elharo@metalab.unc.edu
     </A>

     </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

Atlanta Braves

Batters

Player	P	G	GS	AB	R	H	D	T	HR	RBI	S	CS	SH	SF	E
Marty Malloy	Second Base	11	8	28	3	5	1	0	1	1	0	0	0	0	0
Ozzie Guillen	Shortstop	83	59	264	35	73	15	1	1	22	1	4	4	2	6
Danny Bautista	Outfield	82	27	144	17	36	11	0	3	17	1	0	3	2	2
Gerald Williams	Outfield	129	51	266	46	81	18	3	10	44	11	5	2	1	5
Javier Lopez	Catcher	133	124	489	73	139	21	1	34	106	5	3	1	8	5
Ryan Klesko	Outfield	129	124	427	69	117	29	1	18	70	5	3	0	4	2
Andres Galarraga	First Base	153	151	555	103	169	27	1	44	121	7	6	0	5	11
Wes Helms	Third Base	7	2	13	2	4	1	0	1	2	0	0	0	0	1

Pitchers

Player	P	G	GS	W	L	S	CG	SO	ERA	IP	HR	R	ER	HB	WP	B
Tom Glavine	Starting Pitcher	33	33	20	6	0	4	3	2.47	229.1	13	67	63	2		0

Florida Marlins

Figure 5-9: Pitchers are distinguished from other players after applying the XSL style sheet in Listing 5-7.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 24 of 55

Element Contents and the select Attribute

In this chapter, I focused on using XSL to format data stored in the attributes of an element because it isn't accessible when using CSS. However, XSL works equally well when you want to include an element's character data rather than (or in addition to) its attributes. To indicate that an element's text is to be copied into the output document, simply use the element's name as the value of the `select` attribute of the `xsl:value-of` element. For example, consider, once again, Listing 5-8:

```
Listing 5-8greeting.xml<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="greeting.xsl"?>
<GREETING>
Hello XML!
</GREETING>
```

Let's suppose you want to copy the greeting "Hello XML!" into an `H1` header. First, you use `xsl:for-each` to select the `GREETING` element:

```
<xsl:for-each select="GREETING">
   <H1>
   </H1>
</xsl:for-each>
```

This alone is enough to copy the two `H1` tags into the output. To place the text of the `GREETING` element between them, use `xsl:value-of` with no `select` attribute. Then, by default, the contents of the current element (`GREETING`) are selected. Listing 5-9 shows the complete style sheet.

Listing 5-9: greeting.xsl

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <HTML>
      <BODY>
        <xsl:for-each select="GREETING">
          <H1>
            <xsl:value-of/>
          </H1>
        </xsl:for-each>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```


You can also use `select` to choose the contents of a child element. Simply make the name of the child element the value of the `select` attribute of `xsl:value-of`. For instance, consider the baseball example from the previous chapter in which each player's statistics were stored in child elements rather than in attributes. Given this structure of the document (which is actually far more likely than the attribute-based structure of this chapter) the XSL for the batters' table looks like this:

```
<TABLE>
  <CAPTION><B>Batters</B></CAPTION>
  <THEAD>
   <TR>
    <TH>Player</TH><TH>P</TH><TH>G</TH>
    <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
    <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
    <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
    <TH>E</TH><TH>BB</TH><TH>SO</TH><TH>HBP</TH>
   </TR>
  </THEAD>
 <TBODY>
  <xsl:for-each select="PLAYER[(POSITION
   != 'Starting Pitcher')
   $and$ (POSITION != 'Relief Pitcher')]">
   <TR>
    <TD>
     <xsl:value-of select="GIVEN_NAME"/>
     <xsl:value-of select="SURNAME"/>
    </TD>
    <TD><xsl:value-of select="POSITION"/></TD>
    <TD><xsl:value-of select="GAMES"/></TD>
    <TD>
      <xsl:value-of select="GAMES_STARTED"/>
    </TD>
    <TD><xsl:value-of select="AT_BATS"/></TD>
    <TD><xsl:value-of select="RUNS"/></TD>
    <TD><xsl:value-of select="HITS"/></TD>
    <TD><xsl:value-of select="DOUBLES"/></TD>
    <TD><xsl:value-of select="TRIPLES"/></TD>
    <TD><xsl:value-of select="HOME_RUNS"/></TD>
    <TD><xsl:value-of select="RBI"/></TD>
    <TD><xsl:value-of select="STEALS"/></TD>
    <TD>
     <xsl:value-of select="CAUGHT_STEALING"/>
    </TD>
    <TD>
     <xsl:value-of select="SACRIFICE_HITS"/>
    </TD>
    <TD>
     <xsl:value-of select="SACRIFICE_FLIES"/>
    </TD>
    <TD><xsl:value-of select="ERRORS"/></TD>
```

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 25 of 55


```
<TD><xsl:value-of select="WALKS"/></TD>
<TD>
  <xsl:value-of select="STRUCK_OUT"/>
</TD>
<TD>
  <xsl:value-of select="HIT_BY_PITCH"/>
</TD>
</TR>
</xsl:for-each>  <!-- PLAYER -->
</TBODY>
</TABLE>
```

In this case, within each PLAYER element, the contents of that element's 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 and HIT_BY_PITCH children are extracted and copied to the output. Since we used the same names for the attributes in this chapter as we did for the PLAYER child elements in the last chapter, this example is almost identical to the equivalent section of Listing 5-7. The main difference is that the @ signs are missing. They indicate an attribute rather than a child.

You can do even more with the select attribute. You can select elements: by position (for example, the first, second, last, seventeenth element, and so forth); with particular contents; with specific attribute values; or whose parents or children have certain contents or attribute values. You can even apply a complete set of Boolean logical operators to combine different selection conditions. We will explore more of these possibilities when we return to XSL in Chapter 14.

CSS or XSL?

CSS and XSL overlap to some extent. XSL is certainly more powerful than CSS. However XSL's power is matched by its complexity. This chapter only touched on the basics of what you can do with XSL. XSL is more complicated, and harder to learn and use than CSS, which raises the question, "When should you use CSS and when should you use XSL?"

CSS is more broadly supported than XSL. Parts of CSS Level 1 are supported for HTML elements by Netscape 4 and Internet Explorer 4 (although annoying differences exist). Furthermore, most of CSS Level 1 and some of CSS Level 2 is likely to be well supported by Internet Explorer 5.0 and Mozilla 5.0 for both XML and HTML. Thus, choosing CSS gives you more compatibility with a broader range of browsers.

Additionally, CSS is more stable. CSS level 1 (which covers most of the CSS you've seen so far) and CSS Level 2 are W3C recommendations. XSL is still a very early working draft, and won't be finalized until after this book is printed. Early adopters of XSL have already been burned once, and will be burned again before standards gel. Choosing CSS means you're less likely to have to rewrite your style sheets from month to month just to track evolving software and standards. Eventually, however, XSL will settle down to a usable standard.

Furthermore, since XSL is so new, different software implements different variations and subsets of the draft standard. At the time of this writing (spring 1999) there are at least three major variants of XSL in widespread use. Before this book is published, there will be more. If the incomplete and buggy implementations of CSS in current browsers bother you, the varieties of XSL will drive you insane.

However, XSL is definitely more powerful than CSS. CSS only allows you to apply formatting to element contents. It does not allow you to change or reorder those contents; choose different formatting for elements based on their contents or attributes; or add simple, extra text like a signature block. XSL is far more appropriate when the XML documents contain only the minimum of data and none of the HTML frou-frou that surrounds the data.

With XSL, you can separate the crucial data from everything else on the page, like mastheads, navigation bars, and signatures. With CSS, you have to include all these pieces in your data documents. XML+XSL allows the data documents to live separately from the Web page documents. This makes XML+XSL documents more maintainable and easier to work with.

In the long run XSL should become the preferred choice for real-world, data-intensive applications. CSS is more suitable for simple pages like grandparents use to post pictures of their grandchildren. But for these uses, HTML alone is sufficient. If you've really hit the wall with HTML, XML+CSS doesn't take you much further before you run into another wall. XML+XSL, by contrast, takes you far past the walls of HTML. You still need CSS to work with legacy browsers, but long-term XSL is the way to go.

Summary

In this chapter, you saw examples of creating an XML document from scratch. Specifically, you learned:

- ✦ Information can also be stored in an attribute of an element.
- ✦ An attribute is a name-value pair included in an element's start tag.
- ✦ Attributes typically hold meta-information about the element rather than the element's data.
- ✦ Attributes are less convenient to work with than the contents of an element.

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 26 of 55

✦ Attributes work well for very simple information that's unlikely to change its form as the document evolves. In particular, style and linking information works well as an attribute.

✦ Empty tags offer syntactic sugar for elements with no content.

✦ XSL is a powerful style language that enables you to access and display attribute data and transform documents.

In the next chapter, we'll specify the exact rules that well-formed XML documents must adhere to. We'll also explore some additional means of embedding information in XML documents including comments and processing instructions.

CHAPTER 6

Well-Formed XML Documents

In This Chapter

What XML documents are made of

Markup and character data

Well-formed XML in stand-alone documents

Well-formed HTML

HTML 4.0 has about a hundred different tags. Most of these tags have half a dozen possible attributes for several thousand different possible variations. Because XML is more powerful than HTML, you might think you need to know even more tags, but you don't. XML gets its power through simplicity and extensibility, not through a plethora of tags.

In fact, XML predefines almost no tags at all. Instead XML allows you to define your own tags as needed. However these tags and the documents built from them are not completely arbitrary. Instead they have to follow a specific set of rules which we will elaborate upon in this chapter. A document that follows these rules is said to be *well-formed*. Well-formedness is the minimum criteria necessary for XML processors and browsers to read files. In this chapter, you'll examine the rules for well-formed XML documents and well-formed HTML. Particular attention is paid to how XML differs from HTML.

What XML Documents Are Made Of

An XML document contains text that comprises XML markup and character data. It is a sequential set of bytes of fixed length, which adheres to certain constraints. It may or may not be a file. For instance, an XML document may:

✦ Be stored in a database

✦ Be created on the fly in memory by a CGI program

✦ Be some combination of several different files, each of which is embedded in another

✦ Never exist in a file of its own

CHAPTER 8

Document Type Definitions and Validity

In This Chapter

Document Type Definitions (DTDs)

Document type declarations

Validation against a DTD

The list of elements

Element declarations

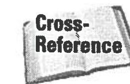
Comments in DTDs

Common DTDs that can be shared among documents

XML 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.



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.

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.

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 &, 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.

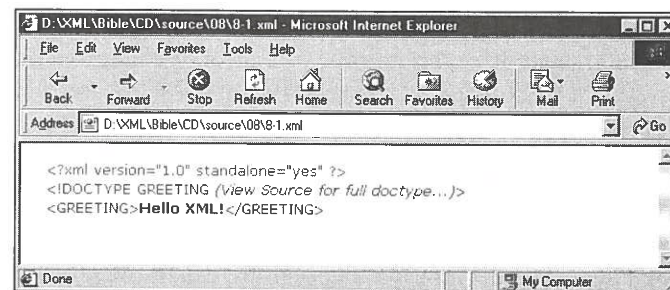


Figure 8-1: Hello XML with DTD displayed in Internet Explorer 5.0

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>
```

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 30 of 55

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/`

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.

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.

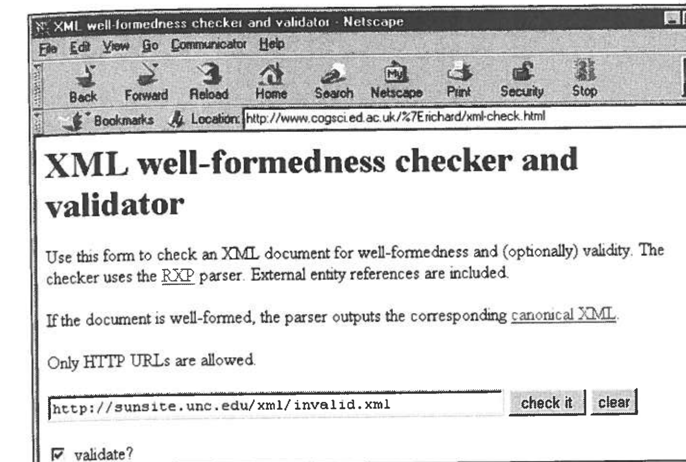


Figure 8-3: Richard Tobin's RXP-based, Web-hosted XML well-formedness checker and validator

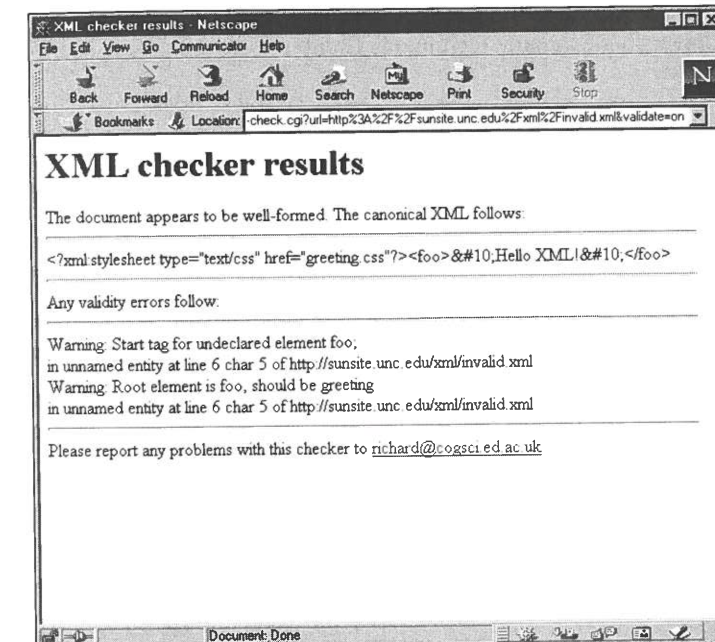


Figure 8-4: The errors with Listing 8-3, as reported by Richard Tobin's XML validator

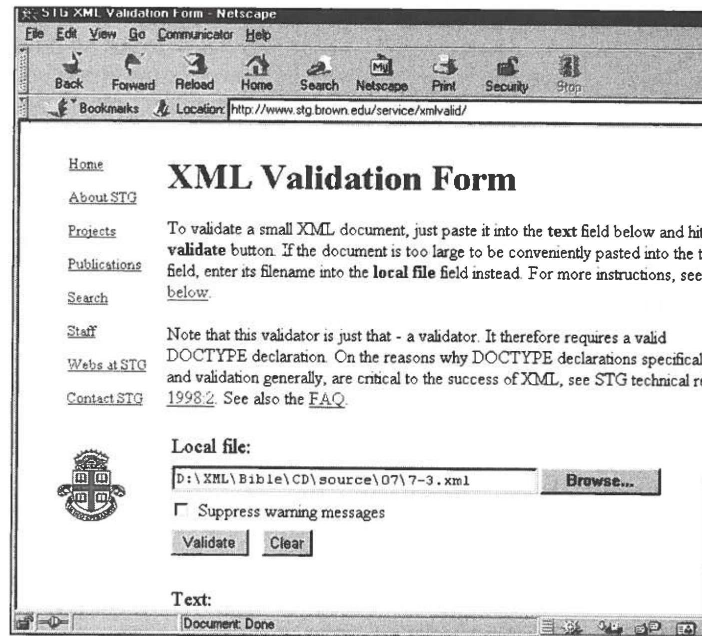


Figure 8-5: Brown University's Scholarly Technology Group's Web-hosted XML validator

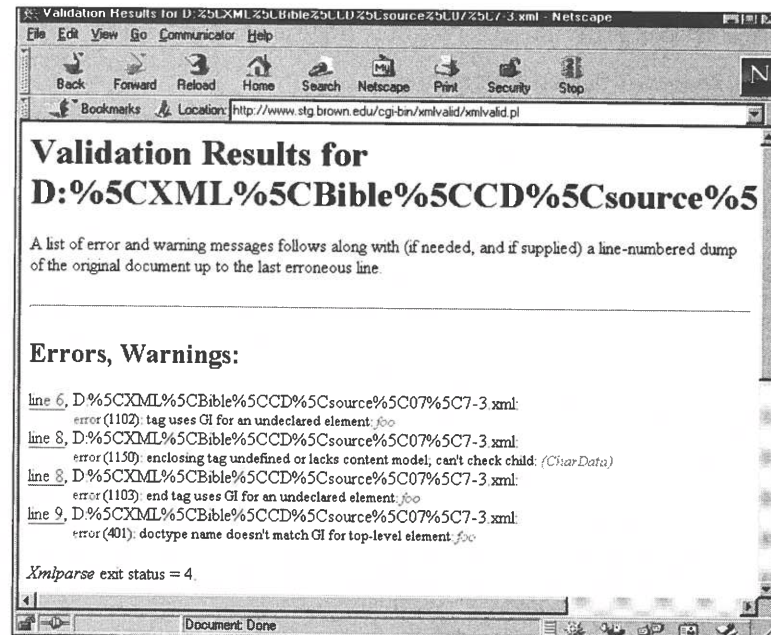


Figure 8-6: The errors with Listing 8-3, as reported by Brown University's Scholarly Technology Group's XML validator

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>
```



```
            <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 example, 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 (s) 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

Element	Elements It Must Contain	Elements It May Contain	Element (if any) in Which It Must Be Contained
SEASON	YEAR,	2 LEAGUE	
YEAR	Text		SEASON
LEAGUE	LEAGUE_NAME, 3 DIVISION		SEASON
LEAGUE_NAME	Text		LEAGUE
DIVISION	DIVISION_NAME , TEAM	TEAM(s)	LEAGUE
DIVISION _NAME	Text		DIVISION
TEAM	TEAM_CITY, TEAM_NAME	PLAYER(s)	DIVISION
TEAM_CITY	Text		TEAM
TEAM_NAME	Text		TEAM
PLAYER	SURNAME, GIVEN _NAME, 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, COMPLETE _GAMES, SHUT_OUTS, ERA, INNINGS, HIT_ BATTER, WILD_ PITCHES, BALK, WALKED_BATTER, STRUCK_OUT_ BATTER	TEAM
SURNAME	Text		PLAYER
GIVEN_NAME	Text		PLAYER
POSITION	Text		PLAYER

Element	Elements It Must Contain	Elements It May Contain	Element (if any) in Which It Must Be Contained
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)*

Element	*Elements It Must Contain*	*Elements It May Contain*	*Element (if any) in Which It Must Be Contained*
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 it's 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

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 38 of 55

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>
```

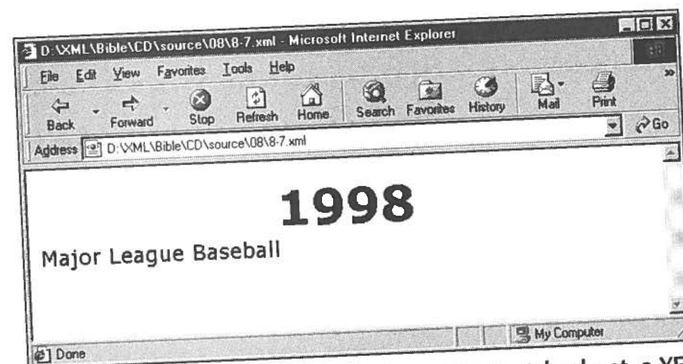


Figure 8-7: A valid document that contains a style sheet, a YEAR element, and normal text displayed in Internet Explorer 5.0

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>
```

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 39 of 55

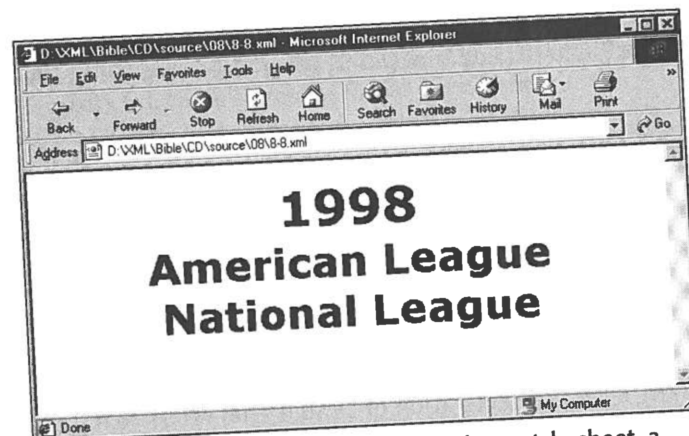


Figure 8-8: A valid document that contains a style sheet, a YEAR element, and two LEAGUE children

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

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 41 of 55

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_CITY>Montreal</TEAM_CITY>
         <TEAM_NAME>Expos</TEAM_NAME>
```

continued

Listing 8-9 *(continued)*

```
        </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 dl element looks like this:

```
<!ELEMENT dl (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>1999 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

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 48 of 55

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">
```

Case 1:19-cv-00859-RTH Document 82-1 Filed 04/29/22 Page 53 of 55

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:

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

This example says this DTD is not standards-body approved (-), belongs to Elliotte 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
  "-//Elliotte 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 it's 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>
```

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.

CHAPTER 9

Entities and External DTD Subsets

In This Chapter

What is an entity?

Internal general entities

External general entities

Internal parameter entities

External parameter entities

How to build a document from pieces

Entities and DTDs in well-formed documents

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

What Is an Entity?

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

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