

# Attribute Declarations in DTDs

---

**S**ome XML elements have attributes. Attributes contain information intended for the application. Attributes are intended for extra information associated with an element (like an ID number) used only by programs that read and write the file, and not for the content of the element that's read and written by humans. In this chapter, you will learn about the various attribute types and how to declare attributes in DTDs.

## What Is an Attribute?

As first discussed in Chapter 3, start tags and empty tags may contain attributes-name-value pairs separated by an equals sign (=). For example,

```
<GREETING LANGUAGE="English">
  Hello XML!
  <MOVIE SOURCE="WavingHand.mov"/>
</GREETING>
```

**In the preceding example, the GREETING element has a LANGUAGE attribute, which has the value English. The MOVIE element has a SOURCE attribute, which has the value WavingHand.mov. The GREETING element's content is Hello XML!. The language in which the content is written is useful information about the content. The language, however, is not itself part of the content.**

**Similarly, the MOVIE element's content is the binary data stored in the file WavingHand.mov. The name of the file is not the content, although the name tells you where the content can be found. Once again, the attribute contains information about the content of the element, rather than the content itself.**



### In This Chapter

What is an attribute?

How to declare attributes in DTDs

How to declare multiple attributes

How to specify default values for attributes

Attribute types

Predefined attributes

A DTD for attribute-based baseball statistics



Elements can possess more than one attribute. For example:

```
<RECTANGLE WIDTH="30" HEIGHT="45"/>
<SCRIPT LANGUAGE="javascript" ENCODING="8859_1">...</SCRIPT>
```

**In this example, the LANGUAGE attribute of the SCRIPT element has the value javascript. The ENCODING attribute of the SCRIPT element has the value 8859\_1. The WIDTH attribute of the RECTANGLE element has the value 30. The HEIGHT attribute of the RECT element has the value 45. These values are all strings, not numbers.**

End tags cannot possess attributes. The following example is illegal:

```
<SCRIPT>...</SCRIPT LANGUAGE="javascript" ENCODING="8859_1">
```

## Declaring Attributes in DTDs

Like elements and entities, the attributes used in a document must be declared in the DTD for the document to be valid. The `<!ATTLIST>` tag declares attributes. `<!ATTLIST>` has the following form:

```
<!ATTLIST Element_name Attribute_name Type Default_value>
```

*Element\_name* is the name of the element possessing this attribute. *Attribute\_name* is the name of the attribute. *Type* is the kind of attribute—one of the ten valid types listed in Table 10-1. The most general type is CDATA. Finally, *Default\_value* is the value the attribute takes on if no value is specified for the attribute.

For example, consider the following element:

```
<GREETING LANGUAGE="Spanish">
  Hola!
</GREETING>
```

This element might be declared as follows in the DTD:

```
<!ELEMENT GREETING (#PCDATA)>
<!ATTLIST GREETING LANGUAGE CDATA "English">
```

The `<!ELEMENT>` tag simply says that a greeting element contains parsed character data. That's nothing new. The `<!ATTLIST>` tag says that GREETING elements have an attribute with the name LANGUAGE whose value has the type CDATA—which is essentially the same as #PCDATA for element content. If you encounter a GREETING tag without a LANGUAGE attribute, the value English is used by default.

**Table 10-1**  
**Attribute Types**

<i>Type</i>	<i>Meaning</i>
CDATA	Character data – text that is not markup
Enumerated	A list of possible values from which exactly one will be chosen
ID	A unique name not shared by any other ID type attribute in the document
IDREF	The value of an ID type attribute of an element in the document
IDREFS	Multiple IDs of elements separated by whitespace
ENTITY	The name of an entity declared in the DTD
ENTITIES	The names of multiple entities declared in the DTD, separated by whitespace
NMTOKEN	An XML name
NOTATION	The name of a notation declared in the DTD
NMTOKENS	Multiple XML names separated by whitespace

The attribute list is declared separately from the tag itself. The name of the element to which the attribute belongs is included in the `<!ATTLIST>` tag. This attribute declaration applies only to that element, which is `GREETING` in the preceding example. If other elements also have `LANGUAGE` attributes, they require separate `<!ATTLIST>` declarations.

As with most declarations, the exact order in which attribute declarations appear is not important. They can come before or after the element declaration with which they're associated. In fact, you can even declare an attribute more than once (though I don't recommend this practice), in which case the first such declaration takes precedence.

You can even declare attributes for tags that don't exist, although this is uncommon. Perhaps you could declare these nonexistent attributes as part of the initial editing of the DTD, with a plan to return later and declare the elements.

## Declaring Multiple Attributes

Elements often have multiple attributes. HTML's `IMG` element can have `HEIGHT`, `WIDTH`, `ALT`, `BORDER`, `ALIGN`, and several other attributes. In fact, most HTML tags

can have multiple attributes. XML tags can also have multiple attributes. For instance, a `RECTANGLE` element naturally needs both a `LENGTH` and a `WIDTH`.

```
<RECTANGLE LENGTH="70px" WIDTH="85px"/>
```

You can declare these attributes in several attribute declarations, with one declaration for each attribute. For example:

```
<!ELEMENT RECTANGLE EMPTY>
<!ATTLIST RECTANGLE LENGTH CDATA "0px">
<!ATTLIST RECTANGLE WIDTH CDATA "0px">
```

The preceding example says that `RECTANGLE` elements possess `LENGTH` and `WIDTH` attributes, each of which has the default value `0px`.

You can combine the two `<!ATTLIST>` tags into a single declaration like this:

```
<!ATTLIST RECTANGLE LENGTH CDATA "0px"
                WIDTH CDATA "0px">
```

This single declaration declares both the `LENGTH` and `WIDTH` attributes, each with type `CDATA` and each with a default value of `0px`. You can also use this syntax when the attributes have different types or defaults, as shown below:

```
<!ATTLIST RECTANGLE LENGTH CDATA "15px"
                WIDTH CDATA "34pt">
```


**Note**

Personally, I'm not very fond of this style. It seems excessively confusing and relies too much on proper placement of extra whitespace for legibility (though the whitespace is unimportant to the actual meaning of the tag). You will certainly encounter this style in DTDs written by other people, however, so you need to understand it.

## Specifying Default Values for Attributes

Instead of specifying an explicit default attribute value like `0px`, an attribute declaration can require the author to provide a value, allow the value to be omitted completely, or even always use the default value. These requirements are specified with the three keywords `#REQUIRED`, `#IMPLIED`, and `#FIXED`, respectively.

### **#REQUIRED**

You may not always have a good option for a default value. For example, when writing a DTD for use on your intranet, you may want to require that all documents

have at least one empty `<AUTHOR/>` tag. This tag is not normally rendered, but it can identify the person who created the document. This tag can have `NAME`, `EMAIL`, and `EXTENSION` attributes so the author may be contacted. For example:

```
<AUTHOR NAME="Elliottte Rusty Harold"
  EMAIL="elharo@metalab.unc.edu" EXTENSION="3459"/>
```

Instead of providing default values for these attributes, suppose you want to force anyone posting a document on the intranet to identify themselves. While XML can't prevent someone from attributing authorship to "Luke Skywalker," it can at least require that authorship is attributed to someone by using `#REQUIRED` as the default value. For example:

```
<!ELEMENT AUTHOR EMPTY>
<!ATTLIST AUTHOR NAME CDATA #REQUIRED>
<!ATTLIST AUTHOR EMAIL CDATA #REQUIRED>
<!ATTLIST AUTHOR EXTENSION CDATA #REQUIRED>
```

If the parser encounters an `<AUTHOR/>` tag that does not include one or more of these attributes, it returns an error.

You might also want to use `#REQUIRED` to force authors to give their `IMG` elements `WIDTH`, `HEIGHT`, and `ALT` attributes. For example:

```
<!ELEMENT IMG EMPTY>
<!ATTLIST IMG ALT CDATA #REQUIRED>
<!ATTLIST IMG WIDTH CDATA #REQUIRED>
<!ATTLIST IMG HEIGHT CDATA #REQUIRED>
```

Any attempt to omit these attributes (as all too many Web pages do) produces an invalid document. The XML processor notices the error and informs the author of the missing attributes.

## #IMPLIED

Sometimes you may not have a good option for a default value, but you do not want to require the author of the document to include a value, either. For example, suppose some of the people posting documents to your intranet are offsite freelancers who have email addresses but lack phone extensions. Therefore, you don't want to require them to include an extension attribute in their `<AUTHOR/>` tags. For example:

```
<AUTHOR NAME="Elliottte Rusty Harold"
  EMAIL="elharo@metalab.unc.edu" />
```

You still don't want to provide a default value for the extension, but you do want to enable authors to include such an attribute. In this case, use `#IMPLIED` as the default value like this:

```
<!ELEMENT AUTHOR EMPTY>
<!ATTLIST AUTHOR NAME      CDATA #REQUIRED>
<!ATTLIST AUTHOR EMAIL     CDATA #REQUIRED>
<!ATTLIST AUTHOR EXTENSION CDATA #IMPLIED>
```

If the XML parser encounters an `<AUTHOR/>` tag without an `EXTENSION` attribute, it informs the XML application that no value is available. The application can act on this notification as it chooses. For example, if the application is feeding elements into a SQL database where the attributes are mapped to fields, the application would probably insert a null into the corresponding database field.

## #FIXED

Finally, you may want to provide a default value for the attribute without allowing the author to change it. For example, you may wish to specify an identical `COMPANY` attribute of the `AUTHOR` element for anyone posting documents to your intranet like this:

```
<AUTHOR NAME="Elliott Rusty Harold" COMPANY="TIC"
  EMAIL="elharo@metalab.unc.edu" EXTENSION="3459"/>
```

You can require that everyone use this value of the company by specifying the default value as `#FIXED`, followed by the actual default. For example:

```
<!ELEMENT AUTHOR EMPTY>
<!ATTLIST AUTHOR NAME      CDATA #REQUIRED>
<!ATTLIST AUTHOR EMAIL     CDATA #REQUIRED>
<!ATTLIST AUTHOR EXTENSION CDATA #IMPLIED>
<!ATTLIST AUTHOR COMPANY   CDATA #FIXED "TIC">
```

Document authors are not required to actually include the fixed attribute in their tags. If they don't include the fixed attribute, the default value will be used. If they do include the fixed attribute, however, they must use an identical value. Otherwise, the parser will return an error.

## Attribute Types

All preceding examples have `CDATA` type attributes. This is the most general type, but there are nine other types permitted for attributes. Altogether the ten types are:

- ♦ CDATA
- ♦ **Enumerated**
- ♦ NMTOKEN
- ♦ NMTOKENS
- ♦ ID
- ♦ IDREF
- ♦ IDREFS
- ♦ ENTITY
- ♦ ENTITIES
- ♦ NOTATION

Nine of the preceding attributes are constants used in the type field, while **Enumerated** is a special type that indicates the attribute must take its value from a list of possible values. Let's investigate each type in depth.

## The CDATA Attribute Type

CDATA, the most general attribute type, means the attribute value may be any string of text not containing a less-than sign (<) or quotation marks ("). These characters may be inserted using the usual entity references (&lt;, and &quot;) or by their Unicode values using character references. Furthermore, all raw ampersands (&)-that is ampersands that do not begin a character or entity reference-must also be escaped as &amp;.

In fact, even if the value itself contains double quotes, they do not have to be escaped. Instead, you may use single quotes to delimit the attributes, as in the following example:

```
<RECTANGLE LENGTH='7"' WIDTH='8.5"' />
```

If the attribute value contains single and double quotes, the one not used to delimit the value must be replaced with the entity references &apos; (apostrophe) and &quot; (double quote). For example:

```
<RECTANGLE LENGTH='8&apos;7"' WIDTH="10'6&quot;"/>
```

## The Enumerated Attribute Type

The enumerated type is not an XML keyword, but a list of possible values for the attribute, separated by vertical bars. Each value must be a valid XML name. The

document author can choose any one member of the list as the value of the attribute. The default value must be one of the values in the list.

For example, suppose you want an element to be visible or invisible. You may want the element to have a `VISIBLE` attribute, which can only have the values `TRUE` or `FALSE`. If that element is the simple `P` element, then the `<!ATTLIST>` declaration would look as follows:

```
<!ATTLIST P VISIBLE (TRUE | FALSE) "TRUE">
```

The preceding declaration says that a `P` element may or may not have a `VISIBLE` attribute. If it does have a `VISIBLE` attribute, the value of that attribute must be either `TRUE` or `FALSE`. If it does not have such an attribute, the value `TRUE` is assumed. For example,

```
<P VISIBLE="FALSE">You can't see me! Nyah! Nyah!</P>
<P VISIBLE="TRUE">You can see me.</P>
<P>You can see me too.</P>
```

By itself, this declaration is not a magic incantation that enables you to hide text. It still relies on the application to understand that it shouldn't display invisible elements. Whether the element is shown or hidden would probably be set through a style sheet rule applied to elements with `VISIBLE` attributes. For example,

```
<xsl:template match="P[@VISIBLE='FALSE']">
</xsl:template>

<xsl:template match="P[@VISIBLE='TRUE']">
  <xsl:apply-templates/>
</xsl:template>
```

## The NMTOKEN Attribute Type

The `NMTOKEN` attribute type restricts the value of the attribute to a valid XML name. As discussed in Chapter 6, XML names must begin with a letter or an underscore (`_`). Subsequent characters in the name may include letters, digits, underscores, hyphens, and periods. They may not include whitespace. (The underscore often substitutes for whitespace.) Technically, names may contain colons, but you shouldn't use this character because it's reserved for use with namespaces.

The `NMTOKEN` attribute type proves useful when you're using a programming language to manipulate the XML data. It's not a coincidence that—except for allowing colons—the preceding rules match the rules for identifiers in Java, JavaScript, and many other programming languages. For example, you could use `NMTOKEN` to associate a particular Java class with an element. Then, you could use Java's reflection API to pass the data to a particular method in a particular class.

The `NMTOKEN` attribute type also helps when you need to pick from any large group of names that aren't specifically part of XML but meet XML's name requirements. The most significant of these requirements is the prohibition of whitespace. For example, `NMTOKEN` could be used for an attribute whose value had to map to an 8.3 DOS file name. On the other hand, it wouldn't work well for UNIX, Macintosh, or Windows NT file-name attributes because those names often contain whitespace.

For example, suppose you want to require a state attribute in an `<ADDRESS/>` tag to be a two-letter abbreviation. You cannot force this characteristic with a DTD, but you can prevent people from entering "New York" or "Puerto Rico" with the following `<!ATTLIST>` declaration:

```
<!ATTLIST ADDRESS STATE NMTOKEN #REQUIRED>
```

However, "California," "Nevada," and other single word states are still legal values. Of course, you could simply use an enumerated list with several dozen two-letter codes, but that approach results in more work than most people want to expend. For that matter, do you even know the two-letter codes for all 50 U.S. states, all the territories and possessions, all foreign military postings, and all Canadian provinces? On the other hand, if you define this list once in a parameter entity reference in a DTD file, you can reuse the file many times over.

## The NMTOKENS Attribute Type

The `NMTOKENS` attribute type is a rare plural form of `NMTOKEN`. It enables the value of the attribute to consist of multiple XML names, separated from each other by whitespace. Generally, you can use `NMTOKENS` for the same reasons as `NMTOKEN`, but only when multiple names are required.

For example, if you want to require multiple two-letter state codes for a state's attribute, you can use the following example:

```
<!ATTLIST ADDRESS STATES NMTOKENS #REQUIRED>
```

Then, you could have an address tag as follows:

```
<ADDRESS STATES="MI NY LA CA">
```

Unfortunately, if you apply this technique, you're no longer ruling out states like New York because each individual part of the state name qualifies as an `NMTOKEN`, as shown below:

```
<ADDRESS STATES="MI New York LA CA">
```

## The ID Attribute Type

An ID type attribute uniquely identifies the element in the document. Authoring tools and other applications commonly use ID to help enumerate the elements of a document without concern for their exact meaning or relationship to one another.

An attribute value of type ID must be a valid XML name—that is, it begins with a letter and is composed of alphanumeric characters and the underscore without whitespace. A particular name may not be used as an ID attribute of more than one tag. Using the same ID twice in one document causes the parser to return an error. Furthermore, each element may not have more than one attribute of type ID.

Typically, ID attributes exist solely for the convenience of programs that manipulate the data. In many cases, multiple elements can be effectively identical except for the value of an ID attribute. If you choose IDs in some predictable fashion, a program can enumerate all the different elements or all the different elements of one type in the document.

The ID type is incompatible with #FIXED. An attribute cannot be both fixed and have ID type because a #FIXED attribute can only have a single value, while each ID type attribute must have a different value. Most ID attributes use #REQUIRED, as Listing 10-1 demonstrates.

### Listing 10-1: A required ID attribute type

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT (P*)>
  <!ELEMENT P (#PCDATA)>
  <!ATTLIST P PNUMBER ID #REQUIRED>
]>

<DOCUMENT>
  <P PNUMBER="p1">The quick brown fox</P>
  <P PNUMBER="p2">The quick brown fox</P>
</DOCUMENT>
```

## The IDREF Attribute Type

The value of an attribute with the IDREF type is the ID of another element in the document. For example, Listing 10-2 shows the IDREF and ID attributes used to connect children to their parents.

**Listing 10-2: family.xml**

```

<?xml version="1.0" standalone="yes"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT (PERSON*)>
  <!ELEMENT PERSON (#PCDATA)>
  <!ATTLIST PERSON PNUMBER ID #REQUIRED>
  <!ATTLIST PERSON FATHER IDREF #IMPLIED>
  <!ATTLIST PERSON MOTHER IDREF #IMPLIED>
]>

<DOCUMENT>
  <PERSON PNUMBER="a1">Susan</PERSON>
  <PERSON PNUMBER="a2">Jack</PERSON>
  <PERSON PNUMBER="a3" MOTHER="a1" FATHER="a2">Chelsea</PERSON>
  <PERSON PNUMBER="a4" MOTHER="a1" FATHER="a2">David</PERSON>
</DOCUMENT>

```

You generally use this uncommon but crucial type when you need to establish connections between elements that aren't reflected in the tree structure of the document. In Listing 10-2, each child is given `FATHER` and `MOTHER` attributes containing the `ID` attributes of its father and mother.

You cannot easily and directly use an `IDREF` to link parents to their children in Listing 10-2 because each parent has an indefinite number of children. As a workaround, you can group all the children of the same parents into a `FAMILY` element and link to the `FAMILY`. Even this approach falters in the face of half-siblings who share only one parent. In short, `IDREF` works for many-to-one relationships, but not for one-to-many relationships.

## The ENTITY Attribute Type

An `ENTITY` type attribute enables you to link external binary data—that is, an external unparsed general entity—into the document. The value of the `ENTITY` attribute is the name of an unparsed general entity declared in the DTD, which links to the external data.

The classic example of an `ENTITY` attribute is an image. The image consists of binary data available from another URL. Provided the XML browser can support it, you may include an image in an XML document with the following declarations in your DTD:

```

<!ELEMENT IMAGE EMPTY>
<!ATTLIST IMAGE SOURCE ENTITY #REQUIRED>
<!ENTITY LOGO SYSTEM "logo.gif">

```

Then, at the desired image location in the document, insert the following `IMAGE` tag:

```
<IMAGE SOURCE="LOGO"/>
```

This approach is not a magic formula that all XML browsers automatically understand. It is simply one technique browsers and other applications may or may not adopt to embed non-XML data in documents.



Cross-Reference

This technique will be explored further in Chapter 11, *Embedding Non-XML Data*.

## The ENTITIES Attribute Type

`ENTITIES` is a relatively rare plural form of `ENTITY`. An `ENTITIES` type attribute has a value part that consists of multiple unparsed entity names separated by whitespace. Each entity name refers to an external non-XML data source. One use for this approach might be a slide show that rotates different pictures, as in the following example:

```
<!ELEMENT SLIDESHOW EMPTY>
<!ATTLIST SLIDESHOW SOURCES ENTITIES #REQUIRED>
<!ENTITY PIC1 SYSTEM "cat.gif">
<!ENTITY PIC2 SYSTEM "dog.gif">
<!ENTITY PIC3 SYSTEM "cow.gif">
```

Then, at the point in the document where you want the slide show to appear, insert the following tag:

```
<SLIDESHOW SOURCES="PIC1 PIC2 PIC3">
```

Once again, this is not a universal formula that all (or even any) XML browsers automatically understand, simply one method browsers and other applications may or may not adopt to embed non-XML data in documents.

## The NOTATION Attribute Type

The `NOTATION` attribute type specifies that an attribute's value is the name of a notation declared in the DTD. The default value of this attribute must also be the name of a notation declared in the DTD. Notations will be introduced in the next chapter. In brief, notations identify the format of non-XML data, for instance by specifying a helper application for an unparsed entity.



Cross-Reference

Chapter 11, *Embedding Non-XML Data*, covers notations.

For example, this `PLAYER` attribute of a `SOUND` element has type `NOTATION`, and a default value of `MP`—the notation signifying a particular kind of sound file:

```
<!ATTLIST SOUND PLAYER NOTATION (MP) #REQUIRED>
<!NOTATION MP SYSTEM "mp1ay32.exe">
```

You can also offer a choice of different notations. One use for this is to specify different helper apps for different platforms. The browser can pick the one it has available. In this case, the `NOTATION` keyword is followed by a set of parentheses containing the list of allowed notation names separated by vertical bars. For example:

```
<!NOTATION MP SYSTEM "mp1ay32.exe">
<!NOTATION ST SYSTEM "soundtool">
<!NOTATION SM SYSTEM "Sound Machine">
<!ATTLIST SOUND PLAYER NOTATION (MP | SM | ST) #REQUIRED>
```

This says that the `PLAYER` attribute of the `SOUND` element may be set to `MP`, `ST`, or `SM`. We'll explore this further in the next chapter.



Note

At first glance, this approach may appear inconsistent with the handling of other list attributes like `ENTITIES` and `NMTOKENS`, but these two approaches are actually quite different. `ENTITIES` and `NMTOKENS` have a list of attributes in the actual element in the document but only one value in the attribute declaration in the DTD. `NOTATION` only has a single value in the attribute of the actual element in the document, however. The list of possible values occurs in the attribute declaration in the DTD.

## Predefined Attributes

In a way, two attributes are predefined in XML. You must declare these attributes in your DTD for each element to which they apply, but you should only use these declared attributes for their intended purposes. Such attributes are identified by a name that begins with `xml:`.

These two attributes are `xml:space` and `xml:lang`. The `xml:space` attribute describes how whitespace is treated in the element. The `xml:lang` attribute describes the language (and optionally, dialect and country) in which the element is written.

### `xml:space`

In HTML, whitespace is relatively insignificant. Although the difference between one space and no space is significant, the difference between one space and two spaces,

one space and a carriage return, or one space, three carriage returns, and 12 tabs is not important. For text in which whitespace is significant—computer source code, certain mainframe database reports, or the poetry of e. e. cummings, for example—you can use a `PRE` element to specify a monospaced font and preservation of whitespace.

XML, however, preserves whitespace by default. The XML processor passes all whitespace characters to the application unchanged. The application usually ignores the extra whitespace. However, the XML processor can tell the application that certain elements contain significant whitespace that should be preserved. The page author uses the `xml:space` attribute to indicate these elements to the application.

If an element contains significant whitespace, the DTD should have an `<!ATTLIST>` for the `xml:space` attribute. This attribute will have an enumerated type with the two values, `default` and `preserve`, as shown in Listing 10-3.

### Listing 10-3: Java source code with significant whitespace encoded in XML

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE PROGRAM [
  <!ELEMENT PROGRAM (#PCDATA)>
  <!ATTLIST PROGRAM xml:space (default|preserve) 'preserve'>
]>
<PROGRAM xml:space="preserve">public class AsciiTable {
    public static void main (String[] args) {
        for (int i = 0; i < 128; i++) {
            System.out.println(i + " " + (char) i);
        }
    }
}
</PROGRAM>
```

---

All whitespace is passed to the application, regardless of whether `xml:space`'s value is `default` or `preserve`. With a value of `default`, however, the application does what it would normally do with extra whitespace. With a value of `preserve`, the application treats the extra whitespace as significant.

Note

Significance depends somewhat on the eventual destination of the data. For instance, extra whitespace in Java source code is relevant to a source code editor but not to a compiler.

**Children of an element for which `xml:space` is defined are assumed to behave similarly as their parent (either preserving or not preserving space), unless they possess an `xml:space` attribute with a conflicting value.**

## xml:lang

The `xml:lang` attribute identifies the language in which the element's content is written. The value of this attribute can have type `CDATA`, `NMTOKEN`, or an enumerated list. Ideally, each of these attributes values should be one of the two-letter language codes defined by the original ISO-639 standard. The complete list of codes can be found on the Web at

<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>.

**For instance, consider the two examples of the following sentence from Petronius's *Satiricon* in both Latin and English. A sentence tag encloses both sentences, but the first sentence tag has an `xml:lang` attribute for Latin while the second has an `xml:lang` attribute for English.**

### Latin:

```
<SENTENCE xml:lang="la">
  Veniebamus in forum deficiente now die, in quo notavimus
  frequentiam rerum venalium, non quidem pretiosarum sed tamen
  quarum fidem male ambulantes obscuritas temporis
  facillime tegeret.
</SENTENCE>
```

### English:

```
<SENTENCE xml:lang="en">
  We have come to the marketplace now when the day is failing,
  where we have seen many things for sale, not for the
  valuable goods but rather that the darkness of
  the time may most easily conceal their shoddiness.
</SENTENCE>
```

**While an English-speaking reader can easily tell which is the original text and which is the translation, a computer can use the hint provided by the `xml:lang` attribute. This distinction enables a spell checker to determine whether to check a particular element and designate which dictionary to use. Search engines can inspect these language attributes to determine whether to index a page and return matches based on the user's preferences.**

## Too Many Languages, Not Enough Codes

XML remains a little behind the times in this area. The original ISO-639 standard language codes were formed from two case-insensitive ASCII alphabetic characters. This standard allows no more than 26 ( 26 or 676 different codes. More than 676 different languages are spoken on Earth today (not even counting dead languages like Etruscan). In practice, the reasonable codes are somewhat fewer than 676 because the language abbreviations should have some relation to the name of the language.

ISO-639, part two, uses three-letter language codes, which should handle all languages spoken on Earth. The XML standard specifically requires two-letter codes, however.

**The language applies to the element and all its children until one of its children declares a different language. The declaration of the SENTENCE element can appear as follows:**

```
<!ELEMENT SENTENCE (#PCDATA)>
<!ATTLIST SENTENCE xml:lang NMTOKEN "en">
```

**If no appropriate ISO code is available, you can use one of the codes registered with the IANA, though currently IANA only adds four additional codes (listed in Table 10-2). You can find the most current list at <http://www.isi.edu/in-notes/iana/assignments/languages/tags>.**

Table 10-2  
The IANA Language Codes

<i>Code</i>	<i>Language</i>
no-bok	Norwegian "Book language"
no-nyn	Norwegian "New Norwegian"
i-navajo	Navajo
i-mingo	Mingo

For example:

```
<P xml:lang="no-nyn">
```

**If neither the ISO nor the IANA has a code for the language you need (Klingon perhaps?), you may define new language codes. These "x-codes" must begin with the string x- or X- to identify them as user-defined, private use codes. For example,**

```
<P xml:lang="x-klingon">
```

The value of the `xml:lang` attribute may include additional subcode segments, separated from the primary language code by a hyphen. Most often, the first subcode segment is a two-letter country code specified by ISO 3166. You can retrieve the most current list of country codes from <http://www.isi.edu/in-notes/iana/assignments/country-codes>. For example:

```
<P xml:lang="en-US">Put the body in the trunk of the car.</P>
<P xml:lang="en-GB">Put the body in the boot of the car.</P>
```

If the first subcode segment does not represent a two-letter ISO country code, it should be a character set subcode for the language registered with the IANA, such as `csDECMCS`, `roman8`, `mac`, `cp037`, or `ebcdic-cp-ca`. The current list can be found at <ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets>. For example:

```
<P xml:lang="en-mac">
```

The final possibility is that the first subcode is another x-code that begins with `x-` or `X-`. For example,

```
<P xml:lang="en-x-tic">
```

By convention, language codes are written in lowercase and country codes are written in uppercase. However, this is merely a convention. This is one of the few parts of XML that is case-insensitive, because of its heritage in the case-insensitive ISO standard.

Like all attributes used in DTDs for valid documents, the `xml:lang` attribute must be specifically declared for those elements to which it directly applies. (It indirectly applies to children of elements that have specified `xml:lang` attributes, but these children do not require separate declaration.)

You may not want to permit arbitrary values for `xml:lang`. The permissible values are also valid XML names, so the attribute is commonly given the `NMTOKEN` type. This type restricts the value of the attribute to a valid XML name. For example,

```
<!ELEMENT P (#PCDATA)>
<!ATTLIST P xml:lang NMTOKEN #IMPLIED "en">
```

Alternately, if only a few languages or dialects are permitted, you can use an enumerated type. For example, the following DTD says that the `P` element may be either English or Latin.

```
<!ELEMENT P (#PCDATA)>
<!ATTLIST P xml:lang (en | la) "en">
```

You can use a `CDATA` type attribute, but there's little reason to. Using `NMTOKEN` or an enumerated type helps catch some potential errors.

## A DTD for Attribute-Based Baseball Statistics

Chapter 5 developed a well-formed XML document for the 1998 Major League Season that used attributes to store the YEAR of a SEASON, the NAME of leagues, divisions, and teams, the CITY where a team plays, and the detailed statistics of individual players. Listing 10-4, below, presents a shorter version of Listing 5-1. It is a complete XML document with two leagues, six divisions, six teams, and two players. It serves to refresh your memory of which elements belong where and with which attributes.

### Listing 10-4: A complete XML document

```
<?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 GIVEN_NAME="Tom" SURNAME="Glavine"
          POSITION="Starting Pitcher" GAMES="33"
          GAMES_STARTED="33" WINS="20" LOSSES="6" SAVES="0"
          COMPLETE_GAMES="4" SHUTOUTS="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" />
      </TEAM>
    </DIVISION>
    <DIVISION NAME="Central">
      <TEAM CITY="Chicago" NAME="Cubs">
        </TEAM>
      </DIVISION>
    <DIVISION NAME="West">
      <TEAM CITY="San Francisco" NAME="Giants">
        </TEAM>
      </DIVISION>
    </LEAGUE>
  <LEAGUE NAME="American League">
    <DIVISION NAME="East">
      <TEAM CITY="New York" NAME="Yankees">
        </TEAM>
      </DIVISION>
    </LEAGUE>
  </SEASON>
</SEASON>
```

```

</DIVISION>
<DIVISION NAME="Central">
  <TEAM CITY="Minnesota" NAME="Twins">
    </TEAM>
  </DIVISION>
<DIVISION NAME="West">
  <TEAM CITY="Oakland" NAME="Athletics">
    </TEAM>
  </DIVISION>
</LEAGUE>
</SEASON>

```

---

In order to make this document valid and well-formed, you need to provide a DTD. This DTD must declare both the elements and the attributes used in Listing 10-4. The element declarations resemble the previous ones, except that there are fewer of them because most of the information has been moved into attributes:

```

<!ELEMENT SEASON (LEAGUE, LEAGUE)>
<!ELEMENT LEAGUE (DIVISION, DIVISION, DIVISION)>
<!ELEMENT DIVISION (TEAM+)>
<!ELEMENT TEAM (PLAYER*)>
<!ELEMENT PLAYER EMPTY>

```

## Declaring SEASON Attributes in the DTD

The SEASON element has a single attribute, YEAR. Although some semantic constraints determine what is and is not a year (1998 is a year; March 31 is not) the DTD doesn't enforce these. Thus, the best approach declares that the YEAR attribute has the most general attribute type, CDATA. Furthermore, we want all seasons to have a year, so we'll make the YEAR attribute required.

```
<!ATTLIST SEASON YEAR CDATA #REQUIRED>
```

Although you really can't restrict the form of the text authors enter in YEAR attributes, you can at least provide a comment that shows what's expected. For example, it may be a good idea to specify that four digit years are required.

```
<!ATTLIST SEASON YEAR CDATA #REQUIRED> <!-- e.g. 1998 -->
<!-- DO NOT USE TWO DIGIT YEARS like 98, 99, 00!! -->
```

## Declaring LEAGUE and DIVISION Attributes in the DTD

Next, consider LEAGUE and DIVISION. Each of these has a single NAME attribute. Again, the natural type is CDATA and the attribute will be required. Since these are

two separate `NAME` attributes for two different elements, two separate `<!ATTLIST>` declarations are required.

```
<!ATTLIST LEAGUE    NAME CDATA #REQUIRED>
<!ATTLIST DIVISION NAME CDATA #REQUIRED>
```

A comment may help here to show document authors the expected form; for instance, whether or not to include the words *League* and *Division* as part of the name.

```
<!ATTLIST LEAGUE    NAME CDATA #REQUIRED>
<!-- e.g. "National League" -->

<!ATTLIST DIVISION NAME CDATA #REQUIRED>
<!-- e.g. "East" -->
```

## Declaring TEAM Attributes in the DTD

A `TEAM` has both a `NAME` and a `CITY`. Each of these is `CDATA` and each is required:

```
<!ATTLIST TEAM NAME CDATA #REQUIRED>
<!ATTLIST TEAM CITY CDATA #REQUIRED>
```

A comment may help to establish what isn't obvious to all; for instance, that the `CITY` attribute may actually be the name of a state in a few cases.

```
<!ATTLIST TEAM NAME CDATA #REQUIRED>
<!ATTLIST TEAM CITY CDATA #REQUIRED>
<!-- e.g. "San Diego" as in "San Diego Padres"
      or "Texas" as in "Texas Rangers" -->
```

Alternately, you can declare both attributes in a single `<!ATTLIST>` declaration:

```
<!ATTLIST TEAM NAME CDATA #REQUIRED
            CITY CDATA #REQUIRED>
```

## Declaring PLAYER Attributes in the DTD

The `PLAYER` element boasts the most attributes. `GIVEN_NAME` and `SURNAME`, the first two, are simply `CDATA` and required:

```
<!ATTLIST PLAYER GIVEN_NAME CDATA #REQUIRED>
<!ATTLIST PLAYER SURNAME    CDATA #REQUIRED>
```

The next `PLAYER` attribute is `POSITION`. Since baseball positions are fairly standard, you might use the enumerated attribute type here. However "First Base," "Second

Base,” “Third Base,” “Starting Pitcher,” and “Relief Pitcher” all contain whitespace and are therefore not valid XML names. Consequently, the only attribute type that works is CDATA. There is no reasonable default value for the position so we make this attribute required as well.

```
<!ATTLIST PLAYER POSITION CDATA #REQUIRED>
```

Next come the various statistics: GAMES, GAMES\_STARTED, AT\_BATS, RUNS, HITS, WINS, LOSSES, SAVES, SHUTOUTS, and so forth. Each should be a number; but as XML has no data typing mechanism, we simply declare them as CDATA. Since not all players have valid values for each of these, let’s declare each one implied rather than required.

```
<!ATTLIST PLAYER GAMES CDATA #IMPLIED>
<!ATTLIST PLAYER GAMES_STARTED CDATA #IMPLIED>

<!-- Batting Statistics -->
<!ATTLIST PLAYER AT_BATS CDATA #IMPLIED>
<!ATTLIST PLAYER RUNS CDATA #IMPLIED>
<!ATTLIST PLAYER HITS CDATA #IMPLIED>
<!ATTLIST PLAYER DOUBLES CDATA #IMPLIED>
<!ATTLIST PLAYER TRIPLES CDATA #IMPLIED>
<!ATTLIST PLAYER HOME_RUNS CDATA #IMPLIED>
<!ATTLIST PLAYER RBI CDATA #IMPLIED>
<!ATTLIST PLAYER STEALS CDATA #IMPLIED>
<!ATTLIST PLAYER CAUGHT_STEALING CDATA #IMPLIED>
<!ATTLIST PLAYER SACRIFICE_HITS CDATA #IMPLIED>
<!ATTLIST PLAYER SACRIFICE_FLIES CDATA #IMPLIED>
<!ATTLIST PLAYER ERRORS CDATA #IMPLIED>
<!ATTLIST PLAYER WALKS CDATA #IMPLIED>
<!ATTLIST PLAYER STRUCK_OUT CDATA #IMPLIED>
<!ATTLIST PLAYER HIT_BY_PITCH CDATA #IMPLIED>

<!-- Pitching Statistics -->
<!ATTLIST PLAYER WINS CDATA #IMPLIED>
<!ATTLIST PLAYER LOSSES CDATA #IMPLIED>
<!ATTLIST PLAYER SAVES CDATA #IMPLIED>
<!ATTLIST PLAYER COMPLETE_GAMES CDATA #IMPLIED>
<!ATTLIST PLAYER SHUTOUTS CDATA #IMPLIED>
<!ATTLIST PLAYER ERA CDATA #IMPLIED>
<!ATTLIST PLAYER INNINGS CDATA #IMPLIED>
<!ATTLIST PLAYER HOME_RUNS_AGAINST CDATA #IMPLIED>
<!ATTLIST PLAYER RUNS_AGAINST CDATA #IMPLIED>
<!ATTLIST PLAYER EARNED_RUNS CDATA #IMPLIED>
<!ATTLIST PLAYER HIT_BATTER CDATA #IMPLIED>
<!ATTLIST PLAYER WILD_PITCHES CDATA #IMPLIED>
<!ATTLIST PLAYER BALK CDATA #IMPLIED>
<!ATTLIST PLAYER WALKED_BATTER CDATA #IMPLIED>
<!ATTLIST PLAYER STRUCK_OUT_BATTER CDATA #IMPLIED>
```

If you prefer, you can combine all the possible attributes of `PLAYER` into one monstrous `<!ATTLIST>` declaration:

```
<!ATTLIST PLAYER
  GIVEN_NAME      CDATA #REQUIRED
  SURNAME         CDATA #REQUIRED
  POSITION         CDATA #REQUIRED
  GAMES          CDATA #IMPLIED
  GAMES_STARTED  CDATA #IMPLIED
  AT_BATS        CDATA #IMPLIED
  RUNS           CDATA #IMPLIED
  HITS           CDATA #IMPLIED
  DOUBLES        CDATA #IMPLIED
  TRIPLES        CDATA #IMPLIED
  HOME_RUNS      CDATA #IMPLIED
  RBI            CDATA #IMPLIED
  STEALS         CDATA #IMPLIED
  CAUGHT_STEALING CDATA #IMPLIED
  SACRIFICE_HITS CDATA #IMPLIED
  SACRIFICE_FLIES CDATA #IMPLIED
  ERRORS         CDATA #IMPLIED
  WALKS          CDATA #IMPLIED
  STRUCK_OUT     CDATA #IMPLIED
  HIT_BY_PITCH   CDATA #IMPLIED

  WINS           CDATA #IMPLIED
  LOSSES         CDATA #IMPLIED
  SAVES          CDATA #IMPLIED
  COMPLETE_GAMES CDATA #IMPLIED
  SHUTOUTS      CDATA #IMPLIED
  ERA           CDATA #IMPLIED
  INNINGS       CDATA #IMPLIED
  HOME_RUNS_AGAINST CDATA #IMPLIED
  RUNS_AGAINST  CDATA #IMPLIED
  EARNED_RUNS   CDATA #IMPLIED
  HIT_BATTER    CDATA #IMPLIED
  WILD_PITCHES  CDATA #IMPLIED
  BALK          CDATA #IMPLIED
  WALKED_BATTER CDATA #IMPLIED
  STRUCK_OUT_BATTER CDATA #IMPLIED>
```

One disadvantage of this approach is that it makes it impossible to include even simple comments next to the individual attributes.

## The Complete DTD for the Baseball Statistics Example

Listing 10-5 shows the complete attribute-based baseball DTD.

**Listing 10-5: The complete DTD for baseball statistics that uses attributes for most of the information**

```

<!ELEMENT SEASON (LEAGUE, LEAGUE)>
<!ELEMENT LEAGUE (DIVISION, DIVISION, DIVISION)>
<!ELEMENT DIVISION (TEAM+)>
<!ELEMENT TEAM (PLAYER*)>
<!ELEMENT PLAYER EMPTY>

<!ATTLIST SEASON    YEAR CDATA #REQUIRED>
<!ATTLIST LEAGUE    NAME CDATA #REQUIRED>
<!ATTLIST DIVISION  NAME CDATA #REQUIRED>
<!ATTLIST TEAM      NAME CDATA #REQUIRED
                  CITY CDATA #REQUIRED>

<!ATTLIST PLAYER    GIVEN_NAME      CDATA #REQUIRED>
<!ATTLIST PLAYER    SURNAME          CDATA #REQUIRED>
<!ATTLIST PLAYER    POSITION          CDATA #REQUIRED>
<!ATTLIST PLAYER    GAMES            CDATA #REQUIRED>
<!ATTLIST PLAYER    GAMES_STARTED   CDATA #REQUIRED>

<!-- Batting Statistics -->
<!ATTLIST PLAYER    AT_BATS          CDATA #IMPLIED>
<!ATTLIST PLAYER    RUNS            CDATA #IMPLIED>
<!ATTLIST PLAYER    HITS            CDATA #IMPLIED>
<!ATTLIST PLAYER    DOUBLES         CDATA #IMPLIED>
<!ATTLIST PLAYER    TRIPLES         CDATA #IMPLIED>
<!ATTLIST PLAYER    HOME_RUNS       CDATA #IMPLIED>
<!ATTLIST PLAYER    RBI             CDATA #IMPLIED>
<!ATTLIST PLAYER    STEALS          CDATA #IMPLIED>
<!ATTLIST PLAYER    CAUGHT_STEALING CDATA #IMPLIED>
<!ATTLIST PLAYER    SACRIFICE_HITS   CDATA #IMPLIED>
<!ATTLIST PLAYER    SACRIFICE_FLIES CDATA #IMPLIED>
<!ATTLIST PLAYER    ERRORS          CDATA #IMPLIED>
<!ATTLIST PLAYER    WALKS           CDATA #IMPLIED>
<!ATTLIST PLAYER    STRUCK_OUT      CDATA #IMPLIED>
<!ATTLIST PLAYER    HIT_BY_PITCH    CDATA #IMPLIED>

<!-- Pitching Statistics -->
<!ATTLIST PLAYER    WINS            CDATA #IMPLIED>
<!ATTLIST PLAYER    LOSSES          CDATA #IMPLIED>
<!ATTLIST PLAYER    SAVES           CDATA #IMPLIED>
<!ATTLIST PLAYER    COMPLETE_GAMES  CDATA #IMPLIED>
<!ATTLIST PLAYER    SHUTOUTS        CDATA #IMPLIED>
<!ATTLIST PLAYER    ERA             CDATA #IMPLIED>
<!ATTLIST PLAYER    INNINGS         CDATA #IMPLIED>
<!ATTLIST PLAYER    HOME_RUNS_AGAINST CDATA #IMPLIED>

```

*Continued*

## Listing 10-5 (continued)

```

<!ATTLIST PLAYER RUNS_AGAINST      CDATA #IMPLIED>
<!ATTLIST PLAYER EARNED_RUNS      CDATA #IMPLIED>
<!ATTLIST PLAYER HIT_BATTER       CDATA #IMPLIED>
<!ATTLIST PLAYER WILD_PITCHES     CDATA #IMPLIED>
<!ATTLIST PLAYER BALK              CDATA #IMPLIED>
<!ATTLIST PLAYER WALKED_BATTER    CDATA #IMPLIED>
<!ATTLIST PLAYER STRUCK_OUT_BATTER CDATA #IMPLIED>

```

To attach the above to Listing 10-4, use the following prolog, assuming of course that Example 10-5 is stored in a file called `baseballattributes.dtd`:

```

<?xml version="1.0" standalone="yes"?>
<!DOCTYPE SEASON SYSTEM "baseballattributes.dtd" >

```

## Summary

In this chapter, you learned how to declare attributes for elements in DTDs. In particular, you learned the following concepts:

- ♦ Attributes are declared in an `<!ATTLIST>` tag in the DTD.
- ♦ One `<!ATTLIST>` tag can declare an indefinite number of attributes for a single element.
- ♦ Attributes normally have default values, but this condition can change by using the keywords `#REQUIRED`, `#IMPLIED`, or `#FIXED`.
- ♦ Ten attribute types can be declared in DTDs: `CDATA`, `Enumerated`, `NMTOKEN`, `NMTOKENS`, `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES`, and `NOTATION`.
- ♦ The predefined `xml:space` attribute determines whether whitespace in an element is significant.
- ♦ The predefined `xml:lang` attribute specifies the language in which an element's content appears.

In the next chapter, you learn how notations, processing instructions, and unparsed external entities can be used to embed non-XML data in XML documents.

