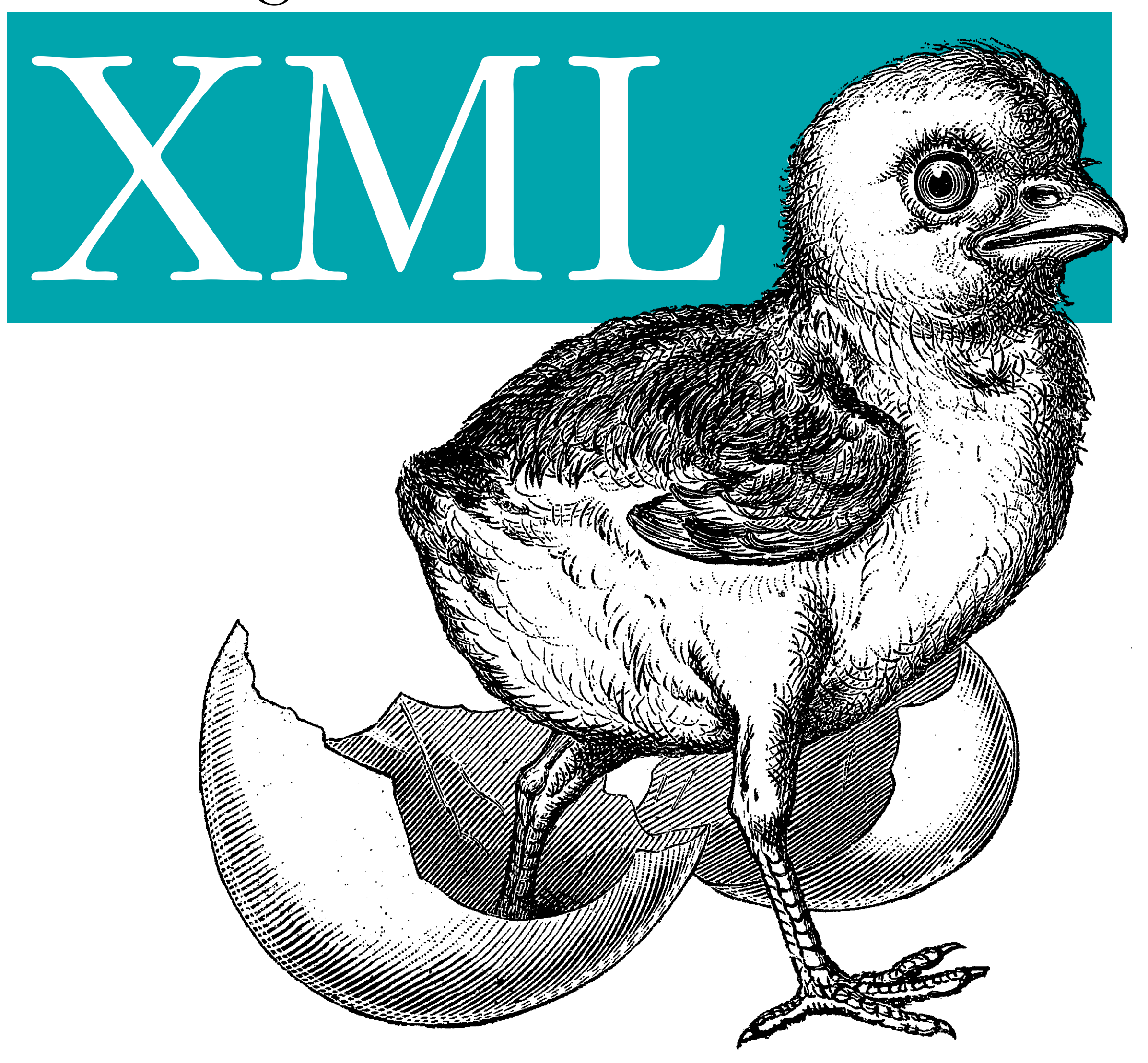
Creating Self-Describing Data

2nd Edition

Covers W3C XML Schema

Learning XML



O'REILLY®

Erik T. Ray

Learning XML

Other XML resources from O'Reilly

Related titles

XML in a Nutshell
XML Pocket Reference
XSLT
XSLT Cookbook
XML Schema
Web Services Essentials
SVG Essentials
Programming Web Services with SOAP
Programming Web Services with XML-RPC
XPath and XPointer
XSL-FO
Perl and XML
Python and XML
Java and XML
Java and XML Data Binding
Java and XSLT

XML Books Resource Center

xml.oreilly.com is a complete catalog of O'Reilly's books on XML and related technologies, including sample chapters and code examples.

XML.com helps you discover XML and learn how this Internet technology can solve real-world problems in information management and electronic commerce.

Conferences

O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

SECOND EDITION

Learning XML

Erik T. Ray

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 2

Markup and Core Concepts

There's a *Far Side* cartoon by Gary Larson about an unusual chicken ranch. Instead of strutting around, pecking at seed, the chickens are all lying on the ground or draped over fences as if they were made of rubber. You see, it was a *boneless* chicken ranch.

Just as skeletons give us vertebrates shape and structure, markup does the same for text. Take out the markup and you have a mess of character data without any form. It would be very difficult to write a computer program that did anything useful with that content. Software relies on markup to label and delineate pieces of data, the way suitcases make it easy for you to carry clothes with you on a trip.

This chapter focuses on the details of XML markup. Here I will describe the fundamental building blocks of all XML-derived languages: elements, attributes, entities, processing instructions, and more. And I'll show you how they all fit together to make a well-formed XML document. Mastering these concepts is essential to understanding every other topic in the book, so read this chapter carefully.

All of the markup rules for XML are laid out in the W3C's technical recommendation for XML version 1.0 (*http://www.w3.org/TR/2000/REC-xml-20001006*). This is the second edition of the original which first appeared in 1998. You may also find Tim Bray's annotated, interactive version useful. Go and check it out at *http://www.xml.com/axml/testaxml.htm*.

Tags

If XML markup is a structural skeleton for a document, then tags are the bones. They mark the boundaries of elements, allow insertion of comments and special instructions, and declare settings for the parsing environment. A parser, the front line of any program that processes XML, relies on tags to help it break down documents into discrete XML objects. There are a handful of different XML object types, listed in Table 2-1.

Table 2-1. Types of tags in XML

Object	Purpose	Example
empty element	Represent information at a specific point in the document.	<xref linkend="abc"/>
container element	Group together elements and character data.	This is a paragraph.
declaration	Add a new parameter, entity, or grammar definition to the parsing environment.	<!ENTITY author "Erik Ray">
processing instruction	Feed a special instruction to a particular type of software.	<?print-formatter force-linebreak?>
comment	Insert an annotation that will be ignored by the XML processor.	<!-- here's where I left off -->
CDATA section	Create a section of character data that should not be parsed, preserving any special characters inside it.	<![CDATA[Ampersands galore! &&&&&&]]>
entity reference	Command the parser to insert some text stored elsewhere.	&company-name;

Elements are the most common XML object type. They break up the document into smaller and smaller cells, nesting inside one another like boxes. Figure 2-1 shows the document in Chapter 1 partitioned into separate elements. Each of these pieces has its own properties and role in a document, so we want to divide them up for separate processing.

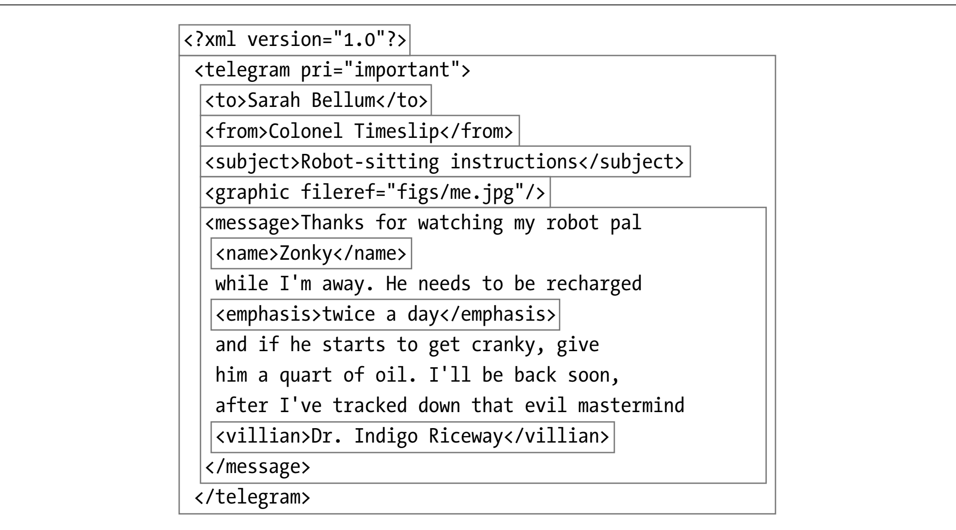


Figure 2-1. Telegram with element boundaries visible

Inside element start tags, you sometimes will see some extra characters next to the element name in the form of `name="value"`. These are *attributes*. They associate

information with an element that may be inappropriate to include as character data. In the telegram example earlier, look for an attribute in the start tag of the `telegram` element.

Declarations are never seen inside elements, but may appear at the top of the document or in an external document type definition file. They are important in setting parameters for the parsing session. They define rules for validation or declare special entities to stand in for text.

The next three objects are used to alter parser behavior while it's going over the document. *Processing instructions* are software-specific directives embedded in the markup for convenience (e.g., storing page numbers for a particular formatter). *Comments* are regions of text that the parser should strip out before processing, as they only have meaning to the author. *CDATA sections* are special regions in which the parser should temporarily suspend its tag recognition.

Rounding out the list are *entity references*, commands that tell the parser to insert predefined pieces of text in the markup. These objects don't follow the pattern of other tags in their appearance. Instead of angle brackets for delimiters, they use the ampersand and semicolon.

In upcoming sections, I'll explain each of these objects in more detail.

Documents

An XML document is a special construct designed to archive data in a way that is most convenient for parsers. It has nothing to do with our traditional concept of documents, like the Magna Carta or *Time* magazine, although those texts could be stored as XML documents. It simply is a way of describing a piece of XML as being whole and intact for parsing.

It's important to think of the document as a *logical* entity rather than a *physical* one. In other words, don't assume that a document will be contained within a single file on a computer. Quite often, a document may be spread out across many files, and some of these may live on different systems. All that is required is that the XML parser reading the document has the ability to assemble the pieces into a coherent whole. Later, we will talk about mechanisms used in XML for linking discrete physical entities into a complete logical unit.

As Figure 2-2 shows, an XML document has two parts. First is the *document prolog*, a special section containing metadata. The second is an element called the *document element*, also called the *root element* for reasons you will understand when we talk about trees. The root element contains all the other elements and content in the document.

The prolog is optional. If you leave it out, the parser will fall back on its default settings. For example, it automatically selects the character encoding UTF-8 (or

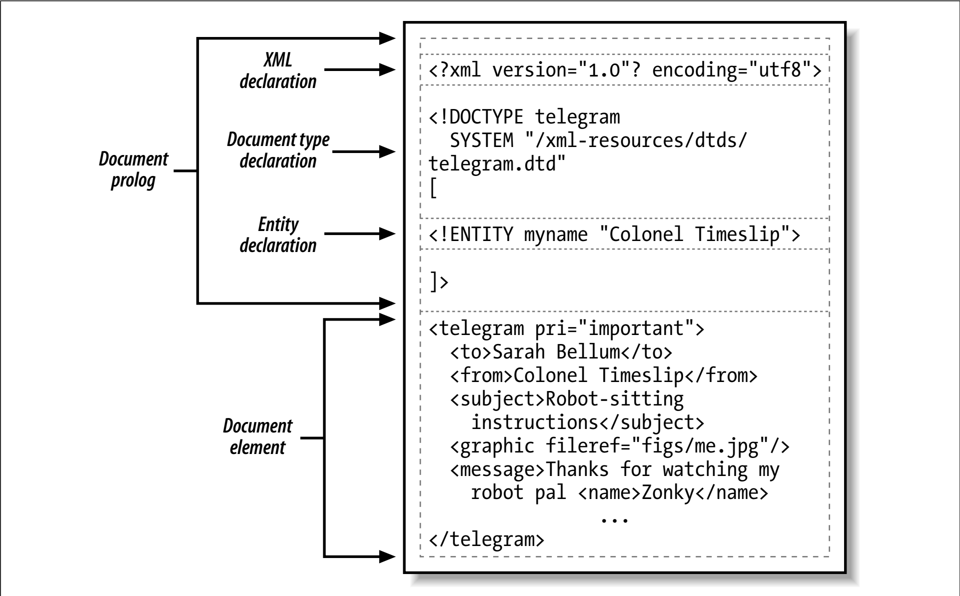


Figure 2-2. Parts of an XML document

UTF-16, if detected) unless something else is specified. The root element is required, because a document without data is just not a document.*

The Document Prolog

Being a flexible markup language toolkit, XML lets you use different character encodings, define your own grammars, and store parts of the document in many places. An XML parser needs to know about these particulars before it can start its work. You communicate these options to the parser through a construct called the document prolog.

The document prolog (if you use one) comes at the top of the document, before the root element. There are two parts (both optional): an XML declaration and a document type declaration.† The first sets parameters for basic XML parsing while the second is for more advanced settings. The XML declaration, if used, has to be the first line in the document. Example 2-1 shows a document containing a full prolog.

* Interestingly, there is no rule that says the root element has to contain anything. This leads to the amusing fact that the following smiley of a perplexed, bearded dunce is a well-formed document: <:-/>. It's an empty element whose name is ":-".

† Don't confuse *document type declaration* with *document type definition*, a completely different beast. To keep the two terms distinct, I will always refer to the latter one with the acronym "DTD."

Example 2-1. A document with a full prolog

```
<?xml version="1.0" standalone="no"?>                The XML declaration
<!DOCTYPE                                            Beginning of the DOCTYPE declaration
  reminder                                           Root element name
  SYSTEM "/home/eray/reminder.dtd"                   DTD identifier
  [                                                  Internal subset start delimiter
    <!ENTITY smile "<graphic file="smile.eps"/>">    Entity declaration
  ]>                                                 Internal subset end delimiter
<reminder>                                           Start of document element
  &smile;                                            Reference to the entity declared above
  <msg>Smile! It can always get worse.</msg>
</reminder>                                          End of document element
```

The XML Declaration

The XML declaration is a small collection of details that prepare an XML processor for working with a document. It is optional, but when used it must always appear in the first line. Figure 2-3 shows the form it takes. It starts with the delimiter `<?xml` (1), contains a number of parameters (2), and ends with the delimiter `?>` (3).

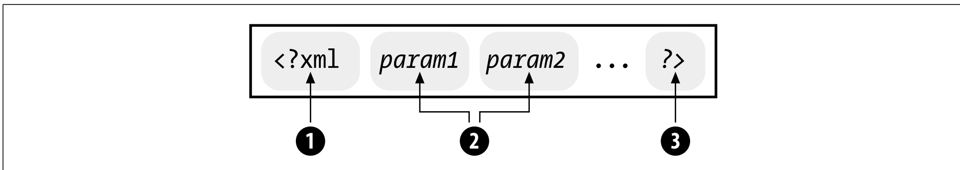


Figure 2-3. Form of the XML declaration

Each parameter consists of a name, an equals sign (=), and a quoted value. The `version` parameter must appear if the other parameters are used:

`version`
: Declares the version of XML used. At the moment, only version 1.0 is officially recognized, but version 1.1 may be available soon.

`encoding`
: Defines the character encoding used in the document. If undefined, the default encoding UTF-8 (or UTF-16, if the document begins with the xFEFF Byte Order Mark) will be used, which works fine for most documents used in English-speaking countries. Character encodings are explained in Chapter 9.

`standalone`
: Informs the parser whether there are any declarations outside of the document. As I explain in the next section, declarations are constructs that contribute information to the parser for assembling and validating a document. The default value is "no"; setting it to "yes" tells the processor there are no external declarations required for parsing the document. It does not, as the name may seem to imply, mean that no other resources need to be loaded. There could well be parts of the document in other files.

Parameter names and values are case-sensitive. The names are always lowercase. Order is important; the version must come before the encoding which must precede the standalone parameter. Either single or double quotes may be used. Here are some examples of XML declarations:

```
<?xml?>
<?xml version="1.0"?>
<?xml version='1.0' encoding='US-ASCII' standalone='yes'?>
<?xml version = '1.0' encoding= 'iso-8859-1' standalone ="no"?>
```

The Document Type Declaration

There are two reasons why you would want to use a document type declaration. The first is to define entities or default attribute values. The second is to support validation, a special mode of parsing that checks grammar and vocabulary of markup. A validating parser needs to read a list of declarations for element rules before it can begin to parse. In both cases, you need to make declarations available, and the place to do that is in the document type declaration section.

Figure 2-4 shows the basic form of the document type declaration. It begins with the delimiter `<!DOCTYPE` (1) and ends with the delimiter `>` (7). Inside, the first part is an element name (2), which identifies the type of the document element. Next is an optional identifier for the document type definition (3), which may be a path to a file on the system, a URL to a file on the Internet, or some other kind of unique name meaningful to the parser. The last part, enclosed in brackets (4 and 6), is an optional list of entity declarations (5) called the *internal subset*. It complements the external document type definition which is called the *external subset*. Together, the internal and external subsets form a collection of declarations necessary for parsing and validation.

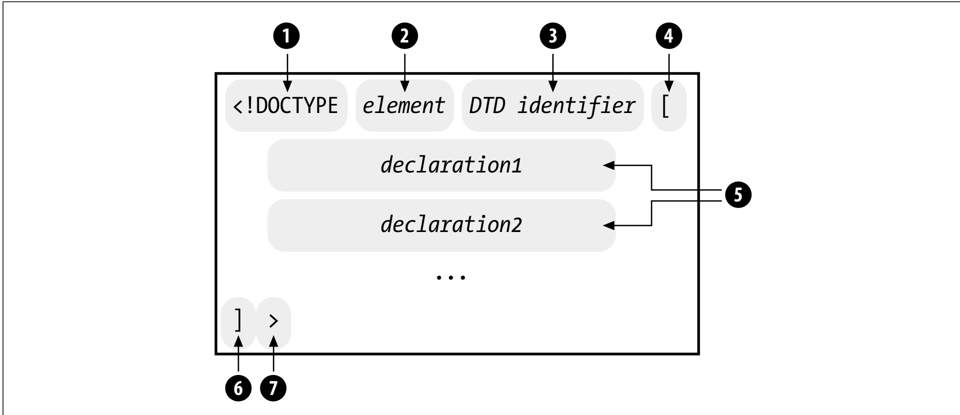


Figure 2-4. Form of the document type declaration

System and public identifiers

The DTD identifier supports two methods of identification: system-specific and public. A *system identifier* takes the form shown in Figure 2-5, the keyword `SYSTEM` (1) followed by a physical address (3) such as a filesystem path or URI, in quotes (2).

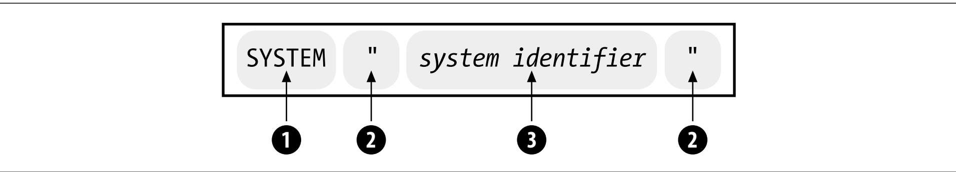


Figure 2-5. Form of the system identifier

Here is an example with a system identifier. It points to a file called *simple.dtd* in the local filesystem.

```
<!DOCTYPE doc
  SYSTEM "/usr/local/xml/dtds/simple.dtd">
```

An alternative scheme to system identifiers is the *public identifier*. Unlike a system path or URI that can change anytime an administrator feels like moving things around, a public identifier is never supposed to change, just as a person may move from one city to another, but her social security number remains the same. The problem is that so far, not many parsers know what to do with public identifiers, and there is no single official registry mapping them to physical locations. For that reason, public identifiers are not considered reliable on their own, and must include an emergency backup system identifier.

Figure 2-6 shows the form of a public identifier. It starts with the keyword `PUBLIC` (1), and follows with a character string (3) in quotes (2), and the backup system identifier (4), also in quotes (2).

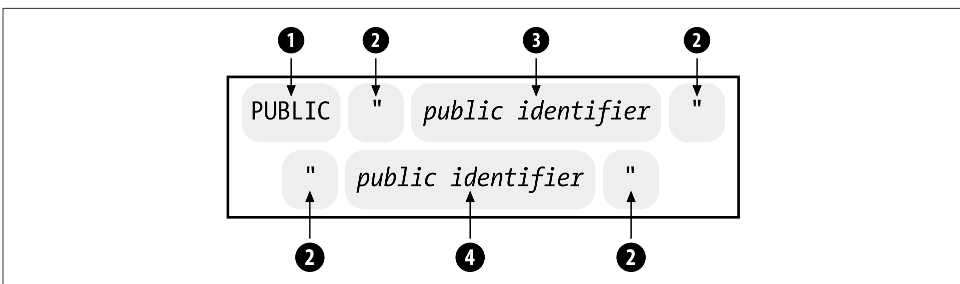


Figure 2-6. Form of the public identifier

Here is an example with a public identifier:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 3.2//EN"
  "http://www.w3.org/TR/HTML/html.dtd">
```

Declarations

Declarations are pieces of information needed to assemble and validate the document. The XML parser first reads declarations from the external subset (given by the system or public identifier), then reads declarations from the internal subset (the portion in square brackets) in the order they appear. In this chapter, I will only talk about what goes in the internal subset, leaving the external subset for Chapter 3.

There are several kinds of declarations. Some have to do with validation, describing what an element may or may not contain (again, I will go over these in Chapter 3). Another kind is the *entity declaration*, which creates a named piece of XML that can be inserted anywhere in the document.

The form of an entity declaration is shown in Figure 2-7. It begins with the delimiter `<!ENTITY` (1), is followed by a name (2), then a value or identifier (3), and the closing delimiter > (4).

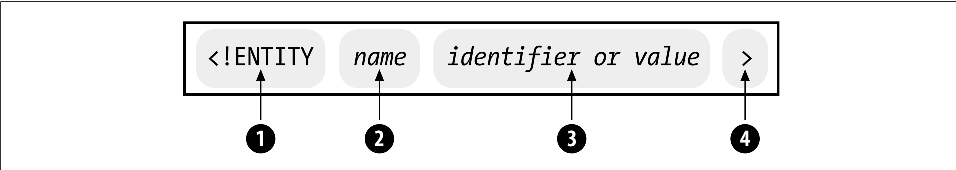


Figure 2-7. Form of an entity declaration

The value or identifier portion may be a system identifier or public identifier, using the same forms shown in Figure 2-5 and Figure 2-6. This associates a name with a piece of XML in a file outside of the document. That segment of XML becomes an *entity*, which is a component of the document that the parser will insert before parsing. For example, this entity declaration creates an entity named `chap2` out of the file *ch02.xml*:

```
<!ENTITY chap2 SYSTEM "ch02.xml">
```

You can insert this entity in the document using an *entity reference* which takes the form in Figure 2-8. It consists of the entity name (2), bounded on the left by an ampersand (1), and on the right by a semicolon (3). You can insert it anywhere in the document element or one of its descendants. The parser will replace it with its value, taken from the external resource, before parsing the document.

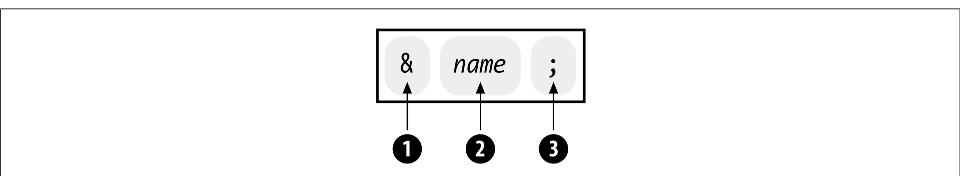


Figure 2-8. Form of an entity reference

In this example, the entity reference is inserted in the XML inside a book element:

```
<book><title>My Exciting Book</title>
&chap2;
</book>
```

Alternatively, an entity declaration may specify an explicit value instead of a system or public identifier. This takes the form of a quoted string. The string can be mixed content (any combination of elements and character data). For example, this declaration creates an entity called jobtitle and assigns it the text <jobtitle>Herder of Cats</jobtitle>:

```
<!ENTITY jobtitle "<jobtitle>Herder of Cats</jobtitle>">
```

We're really just scratching the surface of entities. I'll cover entities in much greater depth later in the chapter.

Elements

Elements are the building blocks of XML, dividing a document into a hierarchy of regions, each serving a specific purpose. Some elements are containers, holding text or elements. Others are empty, marking a place for some special processing such as importing a media object. In this section, I'll describe the rules for how to construct elements.

Syntax

Figure 2-9 shows the syntax for a container element. It begins with a start tag consisting of an angle bracket (1) followed by a name (2). The start tag may contain some attributes (3) separated by whitespace, and it ends with a closing angle bracket (4). After the start tag is the element's content and then an end tag. The end tag consists of an opening angle bracket and a slash (5), the element's name again (2), and a closing bracket (4). The name in the end tag must match the one in the start tag exactly.

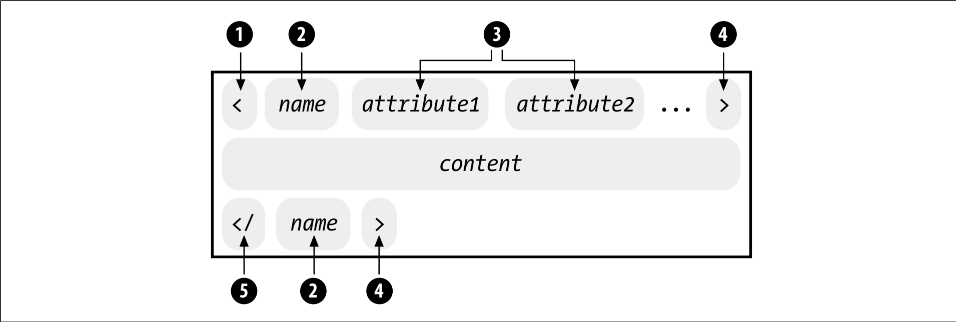


Figure 2-9. Container element syntax

An empty element is very similar, as seen in Figure 2-10. It starts with an angle bracket delimiter (1), and contains a name (2) and a number of attributes (3). It is closed with a slash and a closing angle bracket (4). It has no content, so there is no need for an end tag.

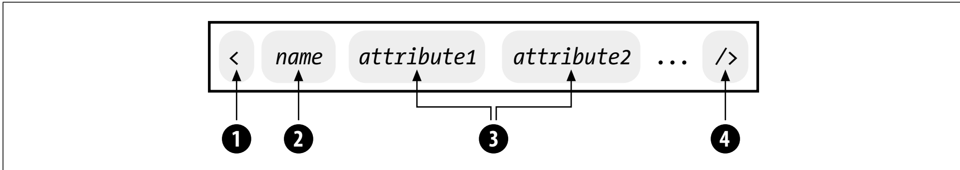


Figure 2-10. Empty element syntax

An attribute defines a property of the element. It associates a name with a value, which is a string of character data. The syntax, shown in Figure 2-11 is a name (1), followed by an equals sign (2), and a string (4) inside quotes (3). Two kinds of quotes are allowed: double (") and single ('). Quote characters around an attribute value must match.

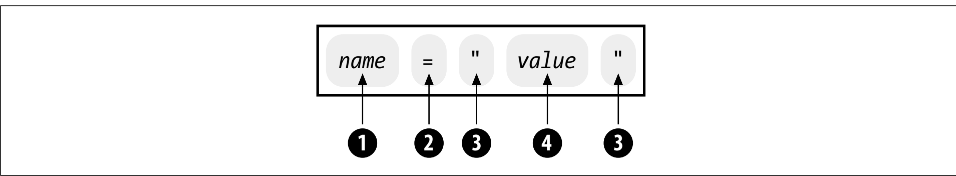


Figure 2-11. Form of an attribute

Element naming must follow the rules of *XML names*, a generic term in the XML specification that also applies to names of attributes and some other kinds of markup. An XML name can contain any alphanumeric characters (a–z, A–Z, and 0–9), accented characters like ç, or characters from non-Latin scripts like Greek, Arabic, or Katakana. The only punctuation allowed in names are the hyphen (-), underscore (_) and period (.). The colon (:) is reserved for another purpose, which I will explain later. Names can only start with a letter, ideograph, or underscore. Names are case-sensitive, so `Para`, `para`, and `pArA` are three different elements.

The following elements are well-formed:

```
<to-do>Clean fish tank</to-do>
<street_address>1420 Sesame Street</street_address>
<MP3.name>Where my doggies at?</MP3.name>
<α3/>
<_-_>goofy, but legal</_-_>
```

These element names are not:

```
<-item>Bathe the badger</-item>
<2nd-phone-number>785-555-1001</2nd-phone-number>
<notes+comments>Huh?</notes+commments>
```

Technically, there is no limit to the length of an XML name. Practically speaking, anything over 50 characters is probably too long.

Inserting whitespace characters (tab, newline, and space) inside the tag is fine, as long as they aren't between the opening angle bracket and the element name. These characters are used to separate attributes. They are also often used to make tags more readable. In the following example, all of the whitespace characters are allowed:

```
<boat
  type="trireme"
><crewmember   class="rower">Dronicus Laborius</crewmember    >
```

There are a few important rules about the tags of container elements. The names in the start and end tags must be identical. An end tag has to come after (never before) the start tag. And both tags have to reside within the same parent element. Violating the last rule is an error called *overlapping*. It's an ambiguous situation where each element seems to contain the other, as you can see here:

```
<a>Don't <b>do</a> this!</b>
```

These untangled elements are okay:

```
<a>No problem</a><b>here</b>
```

Container elements may contain elements or character data or both. Content with both characters and elements is called *mixed content*. For example, here is an element with mixed content:

```
<para>I like to ride my motorcycle
<emphasis>really</emphasis> fast.</para>
```

Attributes

In the element start tag you can add more information about the element in the form of attributes. An *attribute* is a name-value pair. You can use it to add a unique label to an element, place it in a category, add a Boolean flag, or otherwise associate some short string of data. In Chapter 1, I used an attribute in the `telegram` element to set a priority level.

One reason to use attributes is if you want to distinguish between elements of the same name. You don't always want to create a new element for every situation, so an attribute can add a little more granularity in differentiating between elements. In narrative applications like DocBook or HTML, it's common to see attributes like `class` and `role` used for this purpose. For example:

```
<message class="tip">When making crop circles,
push down <emphasis>gently<emphasis> on the stalks to
avoid breaking them.</message>

<message class="warning">Farmers don't like finding people in
their fields at night, so be <emphasis role="bold">very
quiet</emphasis> when making crop circles.</message>
```

The `class` attribute might be used by a stylesheet to specify a special typeface or color. It might format the `<message class="warning">` with a thick border and an icon containing an exclamation point, while the `<message class="tip">` gets an icon of a light bulb and a thin border. The `emphasis` elements are distinguished in whether they have an attribute at all. The second does, and its purpose is to override the default style, whatever that may be.

Another way an attribute can distinguish an element is with a *unique identifier*, a string of characters that is unique to one particular element in the document. No other element may have the same identifier. This gives you a way to select that one element for special treatment, for cross referencing, excerpting, and so on.

For example, suppose you have a catalog with hundreds of product descriptions. Each description is inside a `product` element. You want to create an index of products, with one line per product. How do you refer to a particular `product` among hundreds? The answer is to give each a uniquely identifying label:

```
<product id="display-15-inch-apple">
  ...
</product>
<product id="display-15-inch-sony">
  ...
</product>
<product id="display-15-inch-ibm">
  ...
</product>
```

There is no limit to how many attributes an element can have, as long as no two attributes have the same name. Here's an example of an element start tag with three attributes:

```
<kiosk music="bagpipes" color="red" id="page-81527">
```

This example is not allowed:

```
<!-- Wrong -->
<team person="sue" person="joe" person="jane">
```

To get around this limitation, you could use one attribute to hold all the values:

```
<team persons="sue joe jane">
```

You could also use attributes with different names:

```
<team person1="sue" person2="joe" person3="jane">
```

Or use elements instead:

```
<team>
  <person>sue</person>
  <person>joe</person>
  <person>jane</person>
</team>
```

In a DTD, attributes can be declared to be of certain types. An attribute can have an enumerated value, meaning that the value must be one of a predefined set. Or it may

have a type that registers it as a unique identifier (no other element can have the same value). It may be an identifier reference type, requiring that another element somewhere has an identifier attribute that matches. A validating parser will check all of these attribute types and report deviations from the DTD. I'll have more to say about declaring attribute types in Chapter 4.

Some attribute names are reserved in XML. Typically, they start with the prefix "xml," such as `xmlns`. To avoid a conflict, choose names that don't start with those letters.

Namespaces

Namespaces are a mechanism by which element and attribute names can be assigned to groups. They are most often used when combining different vocabularies in the same document, as I did in Chapter 1. Look at that example, and you'll see attributes in some elements like this one:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
```

Example 2-2 is another case. The `part-catalog` element contains two namespaces which are declared by the attributes `xmlns:nw` and `xmlns`. The elements inside `part-catalog` and their attributes belong to one or the other namespace. Those in the first namespace can be identified by the prefix `nw:`.

Example 2-2. Document with two namespaces

```
<part-catalog
  xmlns:nw="http://www.nutware.com/"
  xmlns="http://www.bobco.com/"
>
  <nw:entry nw:number="1327">
    <nw:description>torque-balancing hexnut</nw:description>
  </nw:entry>
  <part id="555">
    <name>type 4 wingnut</name>
  </part>
</part-catalog>
```

The attributes of `part-catalog` are called *namespace declarations*. The general form of a namespace declaration is illustrated in Figure 2-12. It starts with the keyword `xmlns:` (1) is followed by a *namespace prefix* (2), an equals sign (3), and a *namespace identifier* (5) in quotes (4).

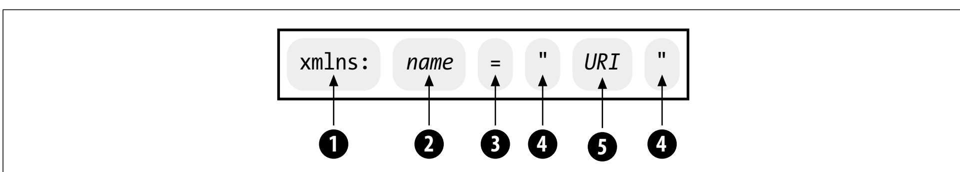
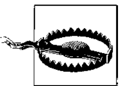


Figure 2-12. Namespace declaration syntax



Avoid using `xml` as a namespace prefix, as it is used in reserved attributes like `xml:space`.

In a special form of the declaration, the colon and namespace prefix are left out, creating an implicit (unnamed) namespace. The second namespace declared in the example above is an implicit namespace. `part-catalog` and any of its descendants without the namespace prefix `nw:` belong to the implicit namespace.

Namespace identifiers are, by convention, assigned to the URL subset of URIs, not the more abstract URNs. This is not a requirement, however. The XML processor doesn't actually look up any information located at that site. The site may not even exist. So why use a URL?

The namespace has to be assigned some kind of unique identifier. URLs are unique. They often contain information about the company or organization. So it makes a good candidate.

Still, many have made the point that URLs are not really *meant* to be used as identifiers. Resources are moved around often, and URLs change. But since no one has found a better method yet, it looks like namespace assignments to URLs is here to stay.

To include an element or attribute in a namespace other than the implicit namespace, you must use the form in Figure 2-13. This is called a *fully qualified name*. To the left of the colon (2) is the namespace prefix (1), and to the right is the *local name* (3).

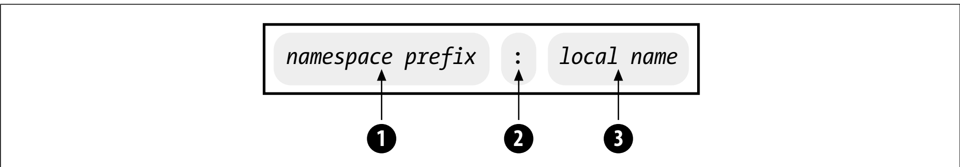


Figure 2-13. Fully qualified name

Namespaces only affect a limited area in the document. The element containing the declaration and all of its descendants are in the scope of the namespace. The element's siblings and ancestors are not. It is also possible to override a namespace by creating another one inside it with the same name. In the following example, there are two namespaces named `flavor`, yet the `chocolate-shell` element is in a different namespace from the element `chewy-center`. The element `flavor:walnut` is in the latter namespace.

```
<flavor:chocolate-shell
  xmlns:flavor="http://www.deliciouscandy.com/chocolate/">
  <flavor:chewy-center
    xmlns:flavor="http://www.deliciouscandy.com/caramel/">
    <flavor:walnut/>
  </flavor:chewy>
</flavor:chocolate-shell>
```

How an XML processor reacts when entering a new namespace depends on the application. For a web document, it may trigger a shift in processing from one kind (e.g., normal web text) to another (e.g., math formulae). Or, as in the case of XSLT, it may use namespaces to sort instructions from data where the former is kind of like a meta-markup.

Namespaces are a wonderful addition to XML, but because they were added after the XML specification, they've created a rather tricky problem. Namespaces do not get along with DTDs. If you want to test the validity of a document that uses non-implicit namespaces, chances are the test will fail. This is because there is no way to write a DTD to allow a document to use namespaces. DTDs want to constrain a document to a fixed set of elements, but namespaces open up documents to an unlimited number of elements. The only way to reconcile the two would be to declare every fully qualified name in the DTD which would not be practical. Until a future version of XML fixes this incompatibility, you will just have to give up validating documents that use multiple namespaces.

Whitespace

You'll notice in my examples, I like to indent elements to clarify the structure of the document. Spaces, tabs, and newlines (collectively called *whitespace* characters) are often used to make a document more readable to the human eye. Take out this visual padding and your eyes will get tired very quickly. So why not add some spaces here and there where it will help?

One important issue is how whitespace should be treated by XML software. At the parser level, whitespace is always passed along with all the other character data to the application level of the program. However, some programs may then *normalize the space*. This process strips out whitespace in element-only content, and in the beginning and end of mixed content. It also collapses a sequence of whitespace characters into a single space.

If you want to prevent a program from removing any whitespace characters from an element, you can give it a hint in the form of the `xml:space` attribute. If you set this attribute to `preserve`, XML processing software is supposed to honor the request by leaving all whitespace characters intact.

Consider this XML-encoded haiku:

```
<poem xml:space="preserve">
A wind shakes the trees,
            An empty sound of sadness.
  The file      is not      here.
</poem>
```

I took some poetic license by putting a bunch of spaces in there. (Hey, it's art!) So how do I keep the XML processor from throwing out the extra space in its normalization process? I gave the `poem` element an attribute named `xml:space`, and set its

value to `preserve`. In Chapter 4, I'll show you how to make this the standard behavior for an element, by making the attribute implicit in the element declaration.

It is not necessary to declare a namespace for `xml:space`. This attribute is built into the XML specification and all XML processors should recognize it.

Some parsers, given a DTD for a document, will make reasonably smart guesses about which elements should preserve whitespace and which should not. Elements that are declared in a DTD to allow mixed content should preserve whitespace, since it may be part of the content. Elements not declared to allow text should have whitespace dropped, since any space in there is only to clarify the markup. However, you can't always rely on a parser to act correctly, so using the `xml:space` attribute is the safest option.

Trees

Elements can be represented graphically as upside-down, tree-like structures. The outermost element, like the trunk of a tree, branches out into smaller elements which in turn branch into other elements until the very innermost content—empty elements and character data—is reached. You can think of the character data as leaves of the tree. Figure 2-14 shows the telegram document drawn as a tree.

Since every XML document has only one possible tree, the diagram acts like a fingerprint, uniquely identifying the document. It's this unambiguous structure that makes XML so useful in containing data. The arboreal metaphor is also useful in thinking about how you would "move" through a document. Documents are parsed from beginning to end, naturally, which happens to correspond to a means of traversing a tree called *depth-first searching*. You start at the root, then move down the first branch to an element, take the first branch from there, and so on to the leaves. Then you backtrack to the last fork and take the next branch, as shown in Figure 2-15.

Let me give you some terminology about XML trees. Every point in a tree—be it an element, text, or something else—is called a *node*. This borrows from graph theory in mathematics, where a tree is a particular type of graph (directed, non-cyclic). Any branch of the tree can be snapped off and thought of as a tree too, just as you can plant the branch of a willow tree to make a new willow tree.* Branches of trees are often called *subtrees* or just trees. Collections of trees are appropriately called *groves*.

An XML tree or subtree (or subsubtree, or subsubsubtree…) must adhere to the rules of well-formedness. In other words, any branch you pluck out of a document could be run through an XML parser, which wouldn't know or care that it wasn't a

* Which is why you should never make fenceposts out of willow wood.

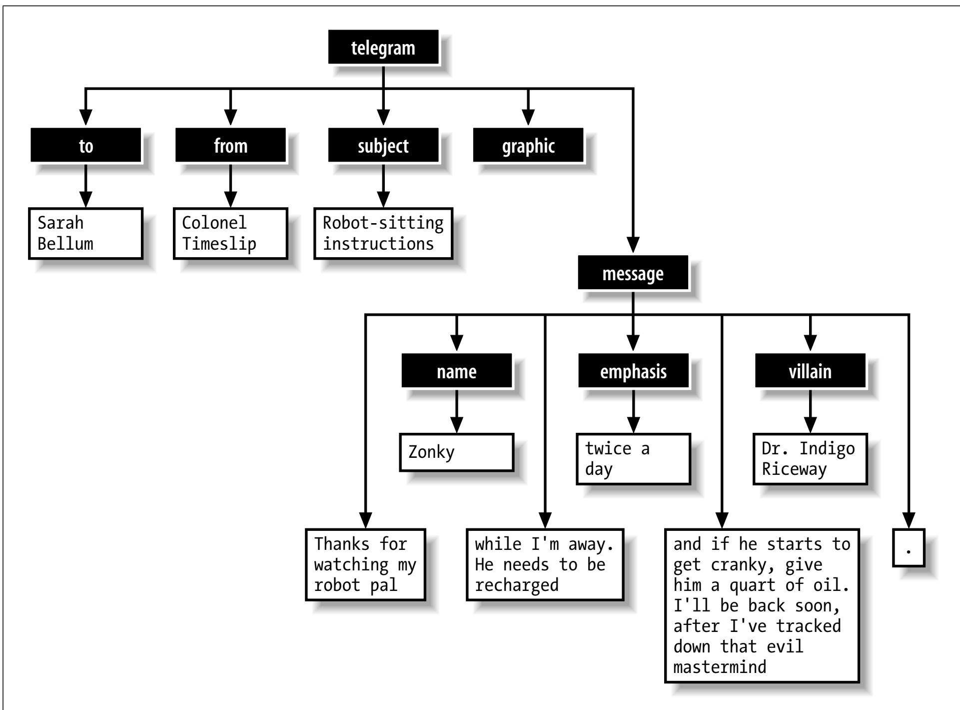


Figure 2-14. A document tree

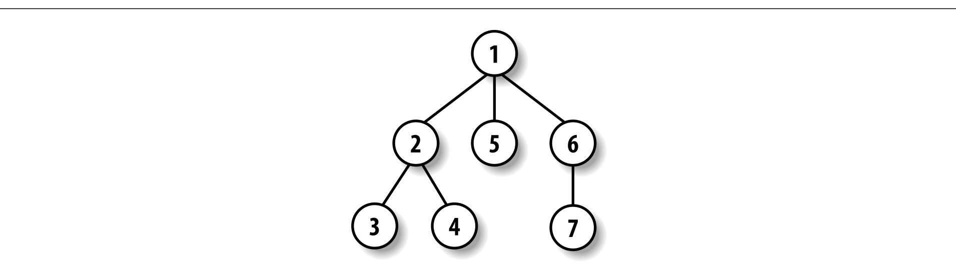


Figure 2-15. Depth-first search

complete document. But a grove (group of adjacent trees) is not well-formed XML. In order to be well-formed, all of the elements must be contained inside just one, the document element.

To describe elements in relation to one another, we use genealogical terms. Imagine that elements are like single-celled organisms, reproducing asexually. You can think of an element as the parent of the nodes it contains, known as its children. So the root of any tree is the progenitor of a whole family with numerous descendants. Likewise, a node may have ancestors and siblings. Siblings to the left (appearing earlier in

the document) are *preceding siblings* while those to the right are *following siblings*. These relationships are illustrated in Figure 2-16.

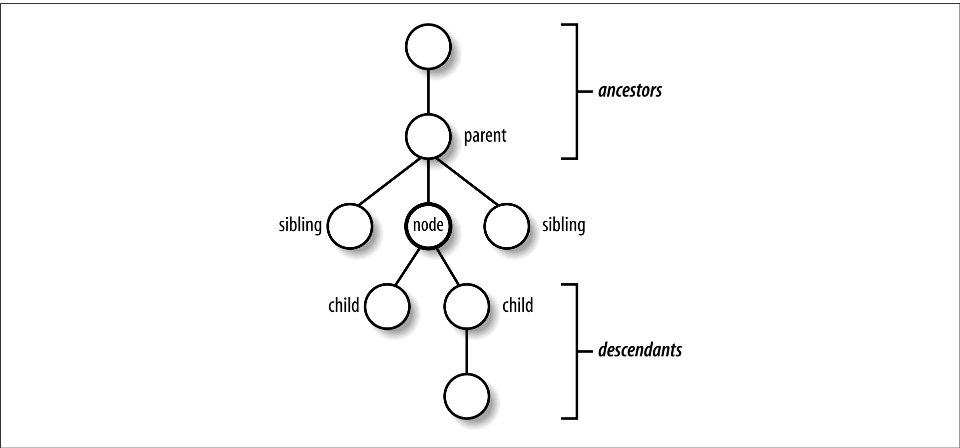


Figure 2-16. Genealogical concepts

The tree model of XML is also important because it represents the way XML is usually stored in computer memory. Each element and region of text is packaged in a cell with pointers to children and parents, and has an object-oriented interface with which to manipulate data. This system is convenient for developers because actions, like moving document parts around and searching for text, are easier and more efficient when separated into tree structures.

Entities

Entities are placeholders in XML. You declare an entity in the document prolog or in a DTD, and you can refer to it many times in the document. Different types of entities have different uses. You can substitute characters that are difficult or impossible to type with character entities. You can pull in content that lives outside of your document with external entities. And rather than type the same thing over and over again, such as boilerplate text, you can instead define your own general entities.

Figure 2-17 shows the different kinds of entities and their roles. In the family tree of entity types, the two major branches are *parameter* entities and *general* entities. *Parameter entities* are used only in DTDs, so I'll talk about them later, in Chapter 4. This section will focus on the other type, general entities.

An entity consists of a name and a value. When an XML parser begins to process a document, it first reads a series of *declarations*, some of which define entities by associating a name with a value. The value is anything from a single character to a file of XML markup. As the parser scans the XML document, it encounters *entity references*, which are special markers derived from entity names. For each entity

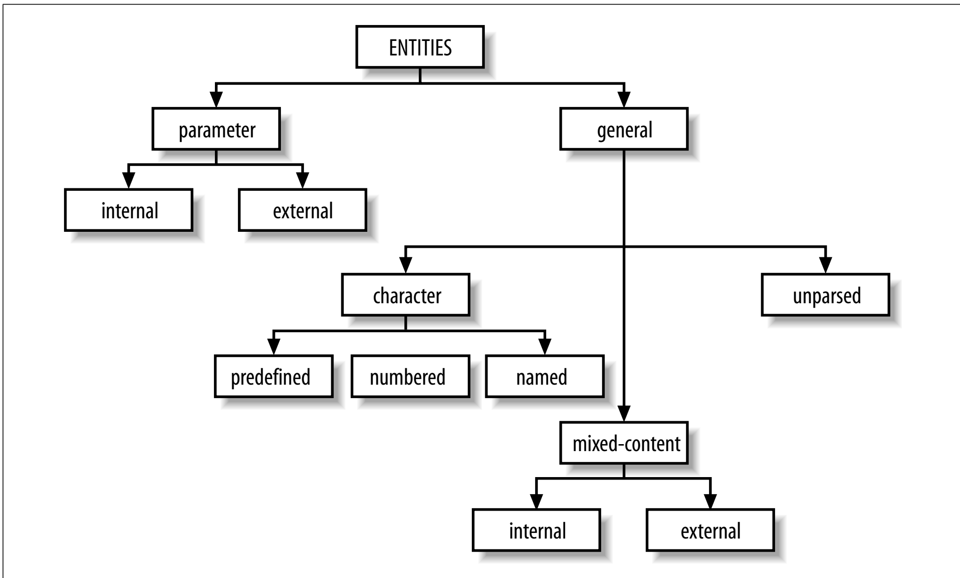


Figure 2-17. Entity types

reference, the parser consults a table in memory for something with which to replace the marker. It replaces the entity reference with the appropriate replacement text or markup, then resumes parsing just before that point, so the new text is parsed too. Any entity references inside the replacement text are also replaced; this process repeats as many times as necessary.

Recall from "The Document Type Declaration" earlier in this chapter that an entity reference consists of an ampersand (&), the entity name, and a semicolon (`;`). The following is an example of a document that declares three general entities and references them in the text:

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "/xmlstuff/dtds/message.dtd"
[
  <!ENTITY client "Mr. Rufus Xavier Sasperilla">
  <!ENTITY agent "Ms. Sally Tashuns">
  <!ENTITY phone "<number>617-555-1299</number>">
]>
<message>
<opening>Dear &client;</opening>
<body>We have an exciting opportunity for you! A set of
ocean-front cliff dwellings in Pi&#241;ata, Mexico, have been
renovated as time-share vacation homes. They're going fast! To
reserve a place for your holiday, call &agent; at &phone;.
Hurry, &client;. Time is running out!</body>
</message>
```

The entities `&client;`, `&agent;`, and `☎` are declared in the internal subset of this document (discussed in "The Document Type Declaration") and referenced in the

<message> element. A fourth entity, `ñ`, is a numbered character entity that represents the character ñ. This entity is referenced but not declared; no declaration is necessary because numbered character entities are implicitly defined in XML as references to characters in the current character set. (For more information about character sets, see Chapter 9.) The XML parser simply replaces the entity with the correct character.

The previous example looks like this with all the entities resolved:

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "/xmlstuff/dtds/message.dtd">
<message>
<opening>Dear Mr. Rufus Xavier Sasperilla</opening>
<body>We have an exciting opportunity for you! A set of
ocean-front cliff dwellings in Piñata, Mexico, have been
renovated as time-share vacation homes. They're going fast! To
reserve a place for your holiday, call Ms. Sally Tashuns at
<number>617-555-1299</number>.
Hurry, Mr. Rufus Xavier Sasperilla. Time is running out!</body>
</message>
```

All entities (besides predefined ones, which I'll describe in a moment) must be declared before they are used in a document. Two acceptable places to declare them are in the internal subset, which is ideal for local entities, and in an external DTD, which is more suitable for entities shared between documents. If the parser runs across an entity reference that hasn't been declared, either implicitly (a predefined entity) or explicitly, it can't insert replacement text in the document because it doesn't know what to replace the entity with. This error prevents the document from being well-formed.

Character Entities

Entities that contain a single character are called, naturally enough, *character entities*. These fall into a few groups:

Predefined character entities
: Some characters cannot be used in the text of an XML document because they conflict with the special markup delimiters. For example, angle brackets (<>) are used to delimit element tags. The XML specification provides the following *predefined character entities*, so you can express these characters safely.

Entity	Value
amp	&
apos	'
gt	>
lt	<
quot	"

Numeric references

XML supports Unicode, a huge character set with tens of thousands of different symbols, letters, and ideograms. You should be able to use any Unicode character in your document. It isn't easy, however, to enter a nonstandard character from a keyboard with less than 100 keys, or to represent one in a text-only editor display. One solution is to use a *numbered character reference* which refers to the character by its number in the Unicode character set.

The number in the entity name can be expressed in decimal or hexadecimal format. Figure 2-18 shows the form of a numeric character entity reference with a decimal number, consisting of the delimiter `&#` (1), the number (2), and a semicolon (3).

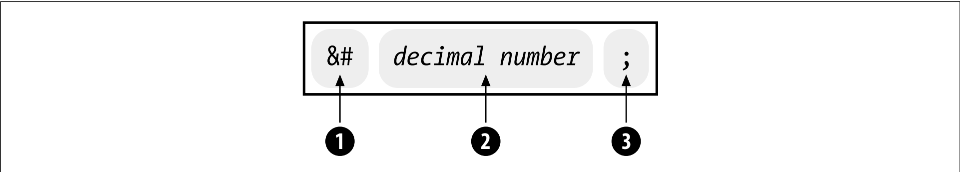


Figure 2-18. Numeric character reference (decimal)

Figure 2-19 shows another form using a hexadecimal number. The difference is that the start delimiter includes the letter "x."

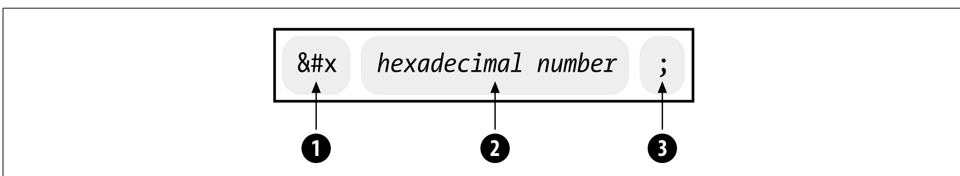


Figure 2-19. Numeric character entity reference (hexadecimal)

For example, a lowercase c with a cedilla (ç) is the 231st Unicode character. It can be represented in decimal as `ç` or in hexadecimal as `ç`. Note that the hexadecimal version is distinguished with an `x` as the prefix to the number. Valid characters are `#x9`, `#xA`, `#xD`, `#x20` through `#xD7FF`, `#xE000` through `#xFFFD`, and `#x10000` through `#x10FFFF`. Since not all hexadecimal numbers map to valid characters, this is not a continuous range. I will discuss character sets and encodings in more detail in Chapter 9.

Named character entities

The problem with numbered character references is that they're hard to remember: you need to consult a table every time you want to use a special character. An easier way to remember them is to use mnemonic entity names. These *named character entities* use easy-to-remember names like `Þ`, which stands for the Icelandic capital thorn character (Þ).

Unlike the predefined and numeric character entities, you do have to declare named character entities. In fact, they are technically no different from other general entities. Nevertheless, it's useful to make the distinction, because large groups of such entities have been declared in DTD modules that you can use in your document. An example is ISO-8879, a standardized set of named character entities including Latin, Greek, Nordic, and Cyrillic scripts, math symbols, and various other useful characters found in European documents.

Mixed-Content Entities

Entity values aren't limited to a single character, of course. The more general *mixed-content entities* have values of unlimited length and can include markup as well as text. These entities fall into two categories: internal and external. For *internal entities*, the replacement text is defined in the entity declaration; for *external entities*, it is located in another file.

Internal entities

Internal mixed-content entities are most often used to stand in for oft-repeated phrases, names, and boilerplate text. Not only is an entity reference easier to type than a long piece of text, but it also improves accuracy and maintainability, since you only have to change an entity once for the effect to appear everywhere. The following example proves this point:

```
<?xml version="1.0"?>
<!DOCTYPE press-release SYSTEM "http://www.dtdland.org/dtds/reports.dtd"
[
  <!ENTITY bobco "Bob's Bolt Bazaar, Inc.">
]>
<press-release>
<title>&bobco; Earnings Report for Q3</title>
<par>The earnings report for &bobco; in fiscal
quarter Q3 is generally good. Sales of &bobco; bolts increased 35%
over this time a year ago.</par>
<par>&bobco; has been supplying high-quality bolts to contractors
for over a century, and &bobco; is recognized as a leader in the
construction-grade metal fastener industry.</par>
</press-release>
```

The entity &bobco; appears in the document five times. If you want to change something about the company name, you only have to enter the change in one place. For example, to make the name appear inside a companyname element, simply edit the entity declaration:

```
<!ENTITY bobco
  "<companyname>Bob's Bolt Bazaar, Inc.</companyname>">
```

When you include markup in entity declarations, be sure not to use the predefined character entities (e.g., < and >) to escape the markup. The parser knows to

read the markup as an entity value because the value is quoted inside the entity declaration. Exceptions to this are the quote-character entity `"` and the single-quote character entity `'`. If they would conflict with the entity declaration's value delimiters, then use the predefined entities, e.g., if your value is in double quotes and you want it to contain a double quote.

Entities can contain entity references, as long as the entities being referenced have been declared previously. Be careful not to include references to the entity being declared, or you'll create a circular pattern that may get the parser stuck in a loop. Some parsers will catch the circular reference, but it is an error.

External entities

Sometimes you may need to create an entity for such a large amount of mixed content that it is impractical to fit it all inside the entity declaration. In this case, you should use an *external entity*, an entity whose replacement text exists in another file. External entities are useful for importing content that is shared by many documents, or that changes too frequently to be stored inside the document. They also make it possible to split a large, monolithic document into smaller pieces that can be edited in tandem and that take up less space in network transfers.

External entities effectively break a document into multiple physical parts. However, all that matters to the XML processor is that the parts assemble into a perfect whole. That is, all the parts in their different locations must still conform to the well-formedness rules. The XML parser stitches up all the pieces into one logical document; with the correct markup, the physical divisions should be irrelevant to the meaning of the document.

External entities are a linking mechanism. They connect parts of a document that may exist on other systems, far across the Internet. The difference from traditional XML links (XLinks) is that for external entities the XML processor must insert the replacement text at the time of parsing.

External entities must always be declared so the parser knows where to find the replacement text. In the following example, a document declares the three external entities `&part1;`, `&part2;`, and `&part3;` to hold its content:

```
<?xml version="1.0"?>
<!DOCTYPE doc SYSTEM "http://www.dtds-r-us.com/generic.dtd"
[
  <!ENTITY part1 SYSTEM "p1.xml">
  <!ENTITY part2 SYSTEM "p2.xml">
  <!ENTITY part3 SYSTEM "p3.xml">
]>
<longdoc>
  &part1;
  &part2;
  &part3;
</longdoc>
```

As shown in Figure 2-20, the file at the top of the pyramid, which we might call the "master file," contains the document declarations and external entity references. The other files are subdocuments—they contain XML, but are not documents in their own right. You could not legally insert document prologs in them. Each may contain more than one XML tree. Though you can't validate them individually (you can only validate a complete document), any errors in a subdocument will affect the whole. External entities don't shield you from parse errors.

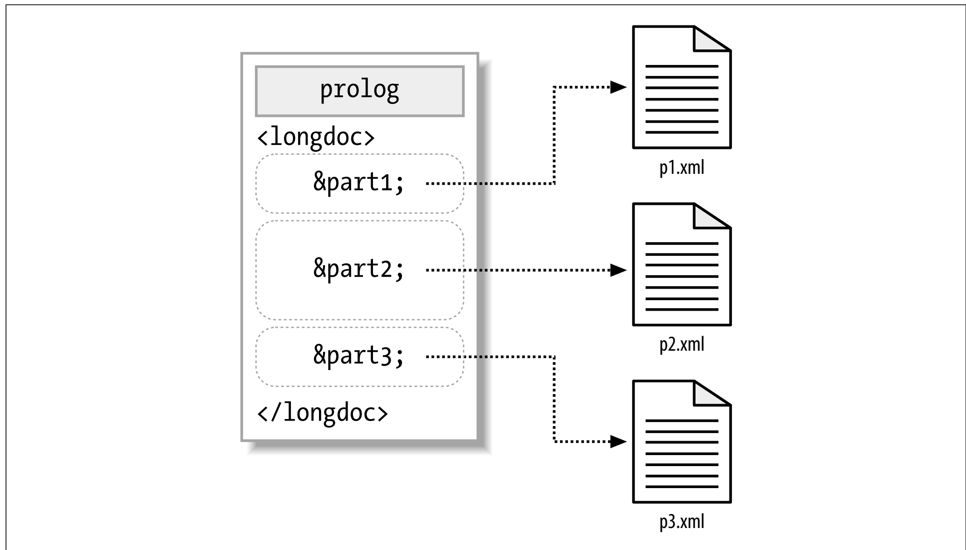


Figure 2-20. Document with external entities

Whenever possible, make each subdocument contain at most one XML tree. While you can't validate a subdocument on its own, you can usually perform a well-formedness check if it has no more than one tree. The parser will think it's looking at a lone document without a prolog. This makes it a lot easier to manage a large document, especially if you have different people working on it at the same time. (This gets tricky if your subdocument uses entities defined in the main document, however.)

The syntax just shown for declaring an external entity uses the keyword `SYSTEM` followed by a quoted string containing a filename. This string is called a *system identifier* and is used to identify a resource by location. The quoted string is actually a URL, so you can include files from anywhere on the Internet. For example:

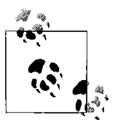
```
<!ENTITY catalog SYSTEM "http://www.bobsbolts.com/catalog.xml">
```

The system identifier suffers from the same drawback as all URLs: if the referenced item is moved, the link breaks. To avoid that problem, you can use a public identifier

in the entity declaration. In theory, a public identifier will endure any location shuffling and still fetch the correct resource. For example:

```
<!ENTITY faraway PUBLIC "-//BOB//FILE Catalog//EN"
    "http://www.bobsbolts.com/catalog.xml">
```

Of course, for this to work, the XML processor has to know how to use public identifiers, and it must be able to find a catalog that maps them to actual locations. In addition, there's no guarantee that the catalog is up to date. A lot can go wrong. Perhaps for this reason, the public identifier must be accompanied by a system identifier (here, `"http://www.bobsbolts.com/catalog.xml"`). If the XML processor for some reason can't handle the public identifier, it falls back on the system identifier. Most web browsers in use today can't deal with public identifiers, so including a backup is a good idea.



The W3C has been working on an alternative to external parsed entities, called XInclude. For details, see *http://www.w3.org/TR/xinclude/*.

Unparsed Entities

The last kind of entity discussed in this chapter is the *unparsed entity*. This kind of entity holds content that should not be parsed because it contains something other than text or XML and would likely confuse the parser. The only place from which unparsed entities can be referred to is in an attribute value. They are used to import graphics, sound files, and other noncharacter data.

The declaration for an unparsed entity looks similar to that of an external entity, with some additional information at the end. For example:

```
<!DOCTYPE doc [
  <!ENTITY mypic SYSTEM "photos/erik.gif" NDATA GIF>
]>
<doc>
  <para>Here's a picture of me:</para>
  <graphic src="&mypic;" />
</doc>
```

This declaration differs from an external entity declaration in that there is an `NDATA` keyword following the system path information. This keyword tells the parser that the entity's content is in a special format, or *notation*, other than the usual parsed mixed content. The `NDATA` keyword is followed by a *notation identifier* that specifies the data format. In this case, the entity is a graphic file encoded in the GIF format, so the word `GIF` is appropriate.

Miscellaneous Markup

Rounding out the list of markup objects are comments, processing instructions, and CDATA sections. They all have one thing in common: they shield content from the parser in some fashion. Comments keep text from ever getting to the parser. CDATA sections turn off the tag resolution, and processing instructions target specific processors.

Comments

Comments are notes in the document that are not interpreted by the XML processor. If you're working with other people on the same files, these messages can be invaluable. They can be used to identify the purpose of files and sections to help navigate a cluttered document, or simply to communicate with each other.

Figure 2-21 shows the form of a comment. It starts with the delimiter `<!--` (1) and ends with the delimiter `-->` (3). Between these delimiters goes the comment text (2) which can be just about any kind of text you want, including spaces, newlines, and markup. The only string not allowed inside a comment is two or more dashes in succession, since the parser would interpret that string as the end of the comment.

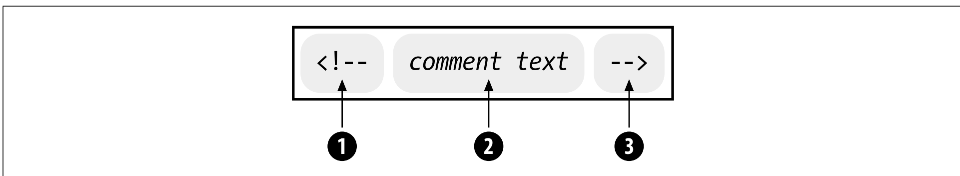


Figure 2-21. Comment syntax

Comments can go anywhere in your document except before the XML declaration and inside tags. The XML processor removes them completely before parsing begins. So this piece of XML:

```
<p>The quick brown fox jumped<!-- test -->over the lazy dog.
The quick brown <!-- test --> fox jumped over the lazy dog. The<!--

test

-->quick brown fox
jumped over the lazy dog.</p>
```

will look like this to the parser:

```
<p>The quick brown fox jumpedover the lazy dog.
The quick brown  fox jumped over the lazy dog. Thequick brown fox
jumped over the lazy dog.</p>
```

Since comments can contain markup, they can be used to "turn off" parts of a document. This is valuable when you want to remove a section temporarily, keeping it in the file for later use. In this example, a region of code is commented out:

```
<p>Our store is located at:</p>
<!--
<address>59 Sunspot Avenue</address>
-->
<address>210 Blather Street</address>
```

When using this technique, be careful not to comment out any comments, i.e., don't put comments inside comments. Since they contain double dashes in their delimiters, the parser will complain when it gets to the inner comment.

CDATA Sections

If you mark up characters frequently in your text, you may find it tedious to use the predefined entities <, >, and &. They require typing and are generally hard to read in the markup. There's another way to include lots of forbidden characters, however: the CDATA section.

CDATA is an acronym for "character data," which just means "not markup." Essentially, you're telling the parser that this section of the document contains no markup and should be treated as regular text. The only thing that cannot go inside a CDATA section is the ending delimiter (]]>).

A CDATA section begins with the nine-character delimiter <![CDATA[(1), and it ends with the delimiter]]> (3). The content of the section (2) may contain markup characters (<, >, and &), but they are ignored by the XML processor (see Figure 2-22).

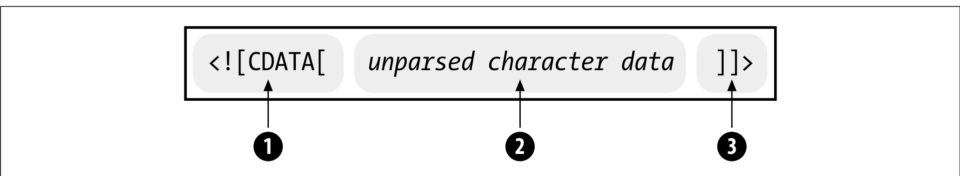


Figure 2-22. CDATA section syntax

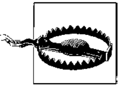
Here's an example of a CDATA section in action:

```
<para>Then you can say "<![CDATA[if (&x < &y)]]>" and be done
with it.</para>
```

This is effectively the same as:

```
<para>Then you can say "if (&amp;x &lt; &amp;y)" and be done
with it.</para>
```

CDATA sections are convenient for large swaths of text that contains a lot of forbidden characters. However, the very thing that makes them useful can also be a problem. You will not be able to use any elements or attributes inside the marked region. If that's a problem for you, then you would probably be better off using character entity references or entities.



You can't nest CDATA sections, because the closing]]> of the nested CDATA section will be treated as the end of the first CDATA section. Because of its role in CDATA sections, you also can't use an unescaped]]> *anywhere* in XML document text.

Processing Instructions

Presentational information should be kept out of a document whenever possible. Still, there may be times when you don't have any other option, for example, if you need to store page numbers in the document to facilitate generation of an index. This information applies only to a specific XML processor and may be irrelevant or misleading to others. The prescription for this kind of information is a *processing instruction*. It is a container for data that is targeted toward a specific XML processor.

Processing instructions (PIs) contain two pieces of information: a target keyword and some data. The parser passes processing instructions up to the next level of processing. If the processing instruction handler recognizes the target keyword, it may choose to use the data; otherwise, the data is discarded. How the data will help processing is up to the developer.

A PI (shown in Figure 2-23) starts with a two-character delimiter <? (1), followed by a *target* (2), an optional string of characters (3) that is the data portion of the PI, and a closing delimiter ?> (4).

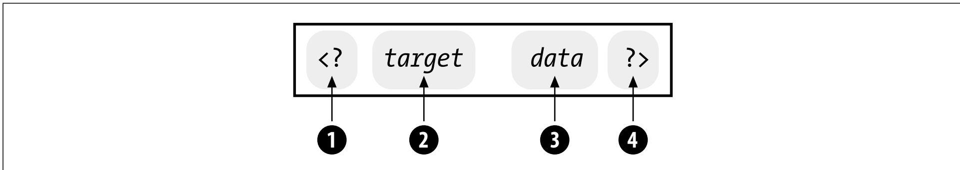


Figure 2-23. Processing instruction syntax

"Funny," you say, "PIs look a lot like the XML declaration." You're right: the XML declaration can be thought of as a processing instruction for all XML processors* that broadcast general information about the document, though the specification defines it as a different thing.

* This syntactic trick allows XML documents to be processed by older SGML systems; they simply treat the XML declaration as another processing instruction, ignoring it since it obviously isn't meant for them.

The target is a keyword that an XML processor uses to determine whether the data is meant for it or not. The keyword doesn't necessarily mean anything, such as the name of the software that will use it. More than one program can use a PI, and a single program can accept multiple PIs. It's sort of like posting a message on a wall saying, "The party has moved to the green house," and people interested in the party will follow the instructions, while those who aren't interested won't.

The PI can contain any data except the combination ?>, which would be interpreted as the closing delimiter. Here are some examples of valid PIs:

```
<?flubber pg=9 recto?>
<?thingie?>
<?xyz stop: the presses?>
```

If there is no data string, the target keyword itself can function as the data. A forced line break is a good example. Imagine that there is a long section heading that extends off the page. Rather than relying on an automatic formatter to break the title just anywhere, we want to force it to break in a specific place.

Here is what a forced line break would look like:

```
<title>The Confabulation of Branklefitzers <?lb?>in a Portlebunky
Frammins <?lb?>Without Denaculization of <?lb?>Crunky Grabblefooties
</title>
```

Now you know all the ins and outs of markup. You can read and understand any XML document as if you were a living XML parser. But it still may not be clear to you *why* things are marked up as they are, or *how* to mark up a bunch of data. In the next chapter, I'll cover these issues as we look at the fascinating topic of data modeling.