Documents

Chapter 2

This chapter describes issues that apply to an XML document as a unit. It also covers the smallest possible units that can be combined to eventually form a document: the characters that a document can use, what we mean by "space", and various declarations and constructs such as comments and processing instructions that don't take an active part in defining a document's logical or physical structure.

A data object is an *XML document* if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.

Splitting XML documents into these two tiers made it easier to achieve design goal five of the spec: that the "number of optional features in XML is to be kept to the absolute minimum, ideally zero". The two-tier system offers flexibility in the ease of document creation and strictness of conformance without requiring sending and receiving systems to run down a checklist of "optional" features that each may or may not support.

In 1.2, "Terminology", entries for "well-formedness constraint" and "validity constraint" tell more about the form these "further constraints" take.

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly, as described in **Section 4.3.2: Well-Formed Parsed Entities**.

A document's physical structure is the relationship of the entities (usually, the files) that make up the document. Within a collection of entities, the main one is the "document entity". A program reading the collection reads that one and, for each reference to another entity it finds in the document entity, it proceeds as if the entity itself had replaced the reference to it (that is, "to cause their inclusion in the document"). These entities can in turn refer to other entities; a diagram of their relationships would show branches fanning out from the central entity to form a tree, which is why we call the document entity the "root".

Logical structure is the schematic structure of the information. Just as a database specialist usually designs a relational database's tables and columns before considering the number and organization of files to use in storing the database, a document designer typically plans out the structure of an XML document's elements before worrying about its physical entity structure.

To be as efficient as possible, a document's physical structure is often customized for the host operating system. This separation of physical from logical design issues allows document collections to have a consistent logical design on different operating systems, allowing the development of more portable documents.

For example, a collection of documents conforming to a single document type might be stored on two different computers, each running a different operating system. The documents would all have the same logical structure, but could have different physical structures on each computer, optimized for the characteristics of that computer's operating system.

For more on declarations, see 2.8, "Prolog and Document Type Declaration"; 2.9, "Standalone Document Declaration"; 3.2, "Element Type Declarations"; 3.3, "Attribute-List Declarations"; 4.2, "Entity Declarations"; 4.3.1, "The Text Declaration"; and 4.7, "Notation Declarations". For more on the other document components listed, see 2.5, "Comments"; 4.1, "Character and Entity References"; and 2.6, "Processing Instructions".

"Nesting" is the containment of one entity or logical structure (usually an element) within another. 4.3.2, "Well-Formed Parsed Entities", further describes the requirements of proper nesting.

2.1. Well-Formed XML Documents

A textual object is a well-formed XML document if:

1. Taken as a whole, it matches the production labeled `document`.
2. It meets all the well-formedness constraints given in this specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is *well-formed*.

Production 1 below, for a **`document`**[†], is number one for a reason: as a parser figures out the structure of your XML document, it recognizes combinations of bigger and bigger pieces (or, in computer sci-

† Throughout these annotations, symbols defined by productions are shown in a bold Courier font.

ence parlance, “nonterminals”) that ultimately result in a prolog, document element, and optional miscellaneous markup (processing instructions, white space, and comments). As this production shows, these are combined into the document itself.

Document

```
[1] document      ::=  prolog element Misc*
```

“Well-formedness constraints” are rules attached to certain productions. These specify that, for the nonterminal defined by the production to be considered well-formed, it must also meet any constraints described under the production. For example, production 39 has the constraint `wfc: Element Type Match` listed on its right and described underneath it. Similarly, “validity constraints” are additional rules that a nonterminal must meet if the document containing it is to qualify as valid.

For a computer, parsing is the process of analyzing text notated in some programming or data description language and determining its components in accordance with the grammar of that language. For an XML processor, this means reading text in “parsed” entities, distinguishing the data content from the markup, and analyzing the markup.

4.3.2, “Well-Formed Parsed Entities”, further describes the concept of a well-formed parsed entity. To summarize, it says that all the elements and entities must nest properly (that is, if one is enclosed by another, the enclosed one ends before the enclosing one) and that elements (and other structures listed in 4.3.2, “Well-Formed Parsed Entities”) can’t begin in one entity and end in another.

Tip *Parsing is the first step in validating, or deciding whether a document meets the rules specified by XML and the document's DTD, so the terms "parsing" and "validating" are often confused. Part of the confusion results from people using the expression "the document doesn't parse" to mean "it's not valid" and "it parses" to mean "it's valid".*

Matching the `document` production implies that:

1. It contains one or more elements.
2. There is exactly one element, called the *root*, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

Elements, described further in Chapter 3, "Logical Structures", are the building blocks of an XML document. Elements have start- and end-tags, and may have character data, other elements, or both between these tags. (Empty elements may use an empty-element tag instead of a start- and end-tag pair.) In an HTML document, an `h2` element usually has only character data between the tags (for example, "`<h2>The Fire Sermon</h2>`"), while the start- and end-tags for the HTML `body` element have other elements between them.

XML elements can also be empty. These can be represented as a start-tag immediately followed by an end-tag or as a shorter alternative known as an empty-element tag, which has a slash before its closing ">". For example, an empty HTML `img` element such as `<img src='whale.gif' align='right'>` could be written as `<img src='whale.gif' align='right'/>` in XML.

Just as a document's entities have a root, so do its elements. If you draw a graph of a document's elements that shows each element branching off into its children (that is, the sub-elements, or elements contained within it) there will be one element containing them all.

This element is the root of the tree picture created by your graph, so we call it the root element of the document. (The term is interchangeable with "document element".) It's not enclosed by any other element, and all the other elements of a document are enclosed within it.

As a consequence of this, for each non-root element `C` in the document, there is one other element `P` in the document such that `C` is in the content of `P`, but is not in the content of any other element that is in the content of `P`. `P` is referred to as the *parent* of `C`, and `C` as a *child* of `P`.

In other words, an element anywhere inside the root element has one and only one parent element. (The `C` and `P` in the spec's example stand for "child" and "parent".) A child element is inside its parent and not inside any of the parent's other child elements.

2.2. Characters

A parsed entity contains *text*, a sequence of characters, which may represent markup or character data. A *character* is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646]. Legal characters are tab, carriage return, line feed, and the legal graphic characters of Unicode and ISO/IEC 10646. The use of "compatibility characters", as defined in section 6.8 of [Unicode], is discouraged.

The ISO/IEC 10646 standard created by a joint commission of the ISO and the International Electrotechnical Commission in 1993 specifies the Universal Multiple-Octet Coded Character Set (UCS).

Let's break down this phrase "Universal Multiple-Octet Coded Character Set". The Universal Character Set is a collection of characters (usually, elements of alphabets, numeric digits, and other characters such as punctuation) that aims to represent all the written languages of the world.

What does it mean to do this with multiple octets? An octet is a grouping of eight bits of information. (On PCs and Macintoshes, a byte is eight bits, but not on all other machines, so it's incorrect to always refer to eight bits as a byte.) An octet can represent 256 different values. This is enough for all the characters on an English-language keyboard and some other miscellaneous ones, but certainly not enough to cover all the characters in all the languages that people want to use when storing documents on computers. Doing this requires multiple octets for each character.

Using two octets per character, you can represent 65,536 different characters; the ISO 10646 version of this is known as UCS-2. Four octets, UCS-4, can represent over two billion different characters (of the 32 bits in the four octets of a UCS-4 character, the first must be "0", leaving over two billion possible combinations of the remaining thirty-one bits).

Unicode is a standard developed by the Unicode Consortium for representing characters with 16 bits. This group of mostly American computer manufacturers is a separate organization from the ISO that has worked closely with them to keep their standard aligned with the UCS-2 subset of ISO 10646. These two standards, in order to remain backward-compatible with existing text files, have the same first 128 characters as the 128 characters in the ASCII character set used by PCs, Macintoshes, and UNIX computers. Therefore, an upper-case "A" is still represented by character 65 and a lower-case "a" by character 97. The XML specification cites both standards because citing only one would imply that XML would follow that one's lead if it ever diverged from the other standard, so identifying the two together encourages them to stay in synch.

Unicode represents some characters more than once, with the extra versions known as "compatibility characters". These are added to ease "round-trip" conversions with (that is, conversions into and then back from) other character set standards. The XML spec doesn't look kindly on these because multiple ways to represent the same character—especially when one way is more efficient and the other is only there as a compromise with other standards—leaves more room for error in a text processing system.

Character Range

```
[2] Char ::=  #x9 | #xA | #xD | [#x20-#xD7FF] |    /* any Unicode character,
              [#xE000-#xFFFD] | [#x10000-#x10FFFF] excluding the surrogate
                                                   blocks, FFFE, and FFFF. */
```

"#x" at the beginning of a number shows that it's written in hexadecimal, or base 16 notation, as opposed to the decimal, "base 10" notation that non-programmers are accustomed to. Hexadecimal notation represents the decimal notation numbers 10 through 15 using the letters A through F, so the decimal numbers 8, 9 10, and 11 would be "#x8", "#x9", "#xA", and "#xB" in hexadecimal. "#x10" represents the decimal number 16, "#x11" is 17, "#x12" is 18, "#xA0" is 160 (as is "#xa0"—the case of alphabetic digits in hexadecimal numbers doesn't matter), "#xA1" is 161, and so on. Production 2 shows that the decimal values 9, 10, 13, 32 – 55295, 57344 – 65533, and 65536 – 1114111 can be used to represent XML characters.

Why use hexadecimal, or in programmer slang, "hex"? Translated to binary, a single hex digit requires four bits, so the eight bits represented by two hex digits will fit into, and completely occupy, a single octet. The use of hexadecimal digits is an efficient compromise between the decimal representation so familiar to humans and the binary representation used by computers. It's also important in the XML world because Unicode refers to each character by its hexadecimal, not decimal value.

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in **Section 4.3.3: Character Encoding in Entities**.

Two different entities may use different encodings, or sets of associations between characters and the bit patterns that represent them in computer storage. All programs that process XML documents must accept the ISO 10646 UTF-8 and UTF-16 encodings. UCS Transformation Format (UTF) 8 and UTF-16 are specific encodings of ISO 10646 characters as sequences of octets.

The "discussion of character encodings" alluded to is 4.3.3, "Character Encoding in Entities", on page 210.

2.3. Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space (#x20) characters, carriage returns, line feeds, or tabs.

By "space (#x20) characters", it means ASCII character 32—the character you type by pressing your keyboard's space bar. ("x20" is the hexadecimal equivalent of 32.)

White Space

```
[3] S    ::=  (#x20 | #x9 | #xD | #xA)+
```

#x9 is the character that you type with your Tab key. Carriage returns and line feeds are two different characters (numbers 13 and 10, or in hex, #xD and #xA) used by different operating systems to represent the end of a line of text; see 2.11, "End-of-Line Handling" for more on these.

Characters are classified for convenience as letters, digits, or other characters. Letters consist of an alphabetic or syllabic base character possibly followed by one or more combining characters, or of an ideographic character. Full definitions of the specific characters in each class are given in **Appendix B: Character Classes**.

"España", the Spanish word for "Spain", has a good example of a letter (n-tilde, or "ñ") that could be coded as the alphabetic base character "n" with the combining character "˜".

An ideographic character, unlike the characters of most Western alphabets, represents an object or idea instead of a particular sound. Appendix B, "Character Classes", lists which characters are considered letters, which are digits, and so forth.

A *Name* is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Names beginning with the string "`xml`", or any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

The concept of a name is important in XML because it's used so often in defining other XML constructs. A token, or terminal, is one of the indivisible units of a document. Tokens are combined according to production rules into nonterminals which are combined into larger nonterminals. These eventually form the most important nonterminal of them all: the XML document, a nonterminal defined by production 1. Element type names, "DOCTYPE", and many other strings of characters used in markup are name tokens. (A "full stop", by the way, is the punctuation character also known as a "period".)

When you make up names to use, such as element type or entity names, don't begin them with the letters "XML" in any combination of upper- and lower-case. (The part in the specification paragraph above with all the parentheses is the regular expression way of saying "XML in any combination of upper- and lower-case". See Chapter 6, "Notation", for more on regular expression syntax.)

The spec prohibits names beginning with "XML" so that when the specification designates names with particular meanings for use in XML (for example, the `xml:space` attribute described in 2.10, "White Space Handling"), there will be no conflict. If your document has an element type named `xmlelement` and some future version of XML creates an `xmlelement` keyword for some special purpose that hadn't been invented when you decided on your element type name, you could have a problem with your document.

NOTE: The colon character within XML names is reserved for experimentation with name spaces. Its meaning is expected to be standardized at some future point, at which point those documents using the colon for experimental purposes may need to be updated. (There is no guarantee that any name-space mechanism adopted for XML will in fact use the colon as a name-space delimiter.) In practice, this means that authors should not use the colon in XML names except as part of name-space experiments, but that XML processors should accept the colon as a name character.

The preceding specification paragraph told us that it was OK to use the colon character (":") in names; this one tells us to avoid it. It's being set aside here for eventual use in solving the namespace problem.

A namespace is a set of unique names. Consider a document type that uses element types and entities declared in two other DTDs.[†] For example, if you're doing a business plan for a restaurant, perhaps `finance.dtd` and `kitchen.dtd` both have element types that you need in your document. What if these two DTDs each declare an element type named `instrument`, and the two declarations for this `instrument` element type are different? Which declaration applies when you want to create a new `instrument` element for your document?

One proposal suggests that you assign a name to each of the two DTDs—for example, "kitchen" and "finance"—and then you could refer to the `<kitchen:instrument>` and `<finance:instrument>`

[†] For more on the use of external subsets for markup declarations, see 2.8, "Prolog and Document Type Declaration".

element types to avoid confusion. Whether the namespace problem is resolved with this syntax or some variation of it, the general plan is to somehow use the colon, so don't use it for something else.

An *Nmtoken* (name token) is any mixture of name characters.

A name token is a slightly relaxed version of a name; as production 7 shows, it's a string of **NameChar** characters. (As production 4 shows, these are letters, numeric digits, the period, hyphen, underscore and colon, plus the **CombiningChar** and **Extender** characters listed in Appendix B, "Character Classes".) Unlike a name (see production 5) a name token doesn't have to begin with a letter, underscore, or colon.

The specification doesn't use name tokens in all the different contexts that it uses names. They're only used for one type of attribute that limits the format of the attribute's values.

For example, declaring an `employee` element type's `phone` attribute with the declaration shown in Example 2.1 would give a document's author a very broad leeway in the allowable phone number values (for example, "(4 0 8) 5 5 5 1 2 1 2" with spaces between the numbers would be legal). However, a declaration like that shown in Example 2.2 would tell a validating XML editor to only allow phone numbers that conformed to production 7, thus preventing spaces and various odd punctuation that you wouldn't want in a phone number. (Note that parentheses are also excluded, so NMTOKEN may not be the most ideal choice for a phone number attribute type.)

Example 2.1: Declaring `employee` element type's `phone` attribute as type CDATA

```
<!ATTLIST employee phone CDATA #REQUIRED>
```

Example 2.2: Declaring `employee` element type's `phone` attribute as type NMTOKEN

```
<!ATTLIST employee phone NMTOKEN #REQUIRED>
```

3.3.1, “Attribute Types”, describes the full choice of types available when declaring attributes.

Names and Tokens

[4]	**NameChar**	`::=`	`Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender`
[5]	**Name**	`::=`	`(Letter | '_' | ':') (NameChar)*`
[6]	**Names**	`::=`	`Name (S Name)*`
[7]	**Nmtoken**	`::=`	`(NameChar)+`
[8]	**Nmtokens**	`::=`	`Nmtoken (S Nmtoken)*`

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (`EntityValue`), the values of attributes (`AttValue`), and external identifiers (`SystemLiteral`). Note that a `SystemLiteral` can be parsed without scanning for markup.

“Any quoted string not containing the quotation mark used as a delimiter” means one of two things:

- A string of characters surrounded by but not containing quotation mark characters (or in programmer slang, double quotes, “like those surrounding this phrase”).
- A string of characters surrounded by but not containing apostrophes (programmer slang: single quotes, ‘like those surrounding this phrase’).

An internal entity consisting of the string “Shantih shantih shantih” could be declared as shown in Examples 2.3 or 2.4.

Example 2.3: Entity replacement text surrounded by double quotes in declaration

```
<!ENTITY sss "Shantih shantih shantih">
```

Example 2.4: Entity replacement text surrounded by single quotes in declaration

```
<!ENTITY sss 'Shantih shantih shantih'>
```

Similarly, either double or single quotes could be used to identify the author's initials in the `chapter` element's attribute specification shown in Example 2.5, or in the DTD file name in the DOCTYPE declaration's external identifier shown in Example 2.6.

Example 2.5: Double quotes used to delimit attribute value

```
<chapter author="TSE">
```

Example 2.6: Single quotes used to delimit system literal in a DOCTYPE declaration

```
<!DOCTYPE harangue SYSTEM 'rant.dtd'>
```

The fact that "a **`SystemLiteral`** can be parsed without scanning for markup" means that a parser will treat as data anything that looks like markup in the system literal. For example, the parser will not treat the second, third, and fourth characters of the system literal in Example 2.7 as a reference to an `a` entity even though the "a" is enclosed by the "&" and ";" characters used to delimit an entity reference.

Example 2.7: Markup characters (& and `;`) that won't be treated as markup because they're in a system literal

```
<!DOCTYPE harangue SYSTEM "r&a;nt.dtd">
```

(Don't try this at home with important data—using such punctuation in file names is asking for trouble on any operating system, even if the XML application software can handle it.) 4.2.2, “External Entities”, has more on using system literals in external entity references.

Literals

```
[9] EntityValue      ::=  '"' ([^%&"] | PEReference | Reference)* '"'
                          |  "'" ([^%&'] | PEReference | Reference)* "'"
[10]AttValue         ::=  '"' ([^<&"] | Reference)* '"'
                          |  "'" ([^<&'] | Reference)* "'"
[11]SystemLiteral    ::=  ('"' [^"]* '"') | ("'" [^']* "'")
[12]PubidLiteral     ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13]PubidChar        ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-
                          '()+,./:=?;!*#@$_%]
```

Productions 9 through 13 show delimited strings, or “literals”. The first four productions each offer two nearly identical choices that differ only in whether double or single quotes are the delimiters. The square brackets at the beginning of each expression show which characters are prohibited there. The “`^`” character signifies negation; for example, `[^xyz]` means “any character except `x`, `y`, or `z`”. See Chapter 6, “Notation”, for more on regular expression syntax.

2.4. Character Data and Markup

Text consists of intermingled character data and markup. *Markup* takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions.

All text that is not markup constitutes the *character data* of the document.

After the first specification paragraph above describes the categories of possible markup, the second defines “character data” as being everything else. To start with a simple case, in Example 2.8 the `<h2>` start-tag and the `</h2>` end-tag are the markup and “The Fire Sermon” is the character data.

Example 2.8: Markup plus character data

```
<h2>The Fire Sermon</h2>
```

Other categories of markup listed in the first paragraph:

- *Empty-element tags* such as `<img src="whale.jpg"/>`.
- *Entity references* such as `&lt;` or `&sss;` and *character references* such as `&#60;` `&#x3c;`. 4.1, “Character and Entity References”, covers both of these in more detail.
- *Comment declarations* such as `<!-- check date -->` that the parser may or may not pass on to the application. See 2.5, “Comments”, for more on these.
- *CDATA sections*, which have nothing but character data. If a parser sees “<ingredient>” in a CDATA section, it won’t treat it as the start-tag of an `ingredient` element; it just treats it as the data characters “<ingredient>”. “&lt;” in a CDATA section is not a reference to an `lt` entity; it’s just the data “&lt;”. 2.7, “CDATA Sections”, describes this further.
- A *document type declaration* at the beginning of a document identifies the document’s type and also contains and/or tells where to find its element, attribute, entity, and notation declarations. For example, the document type declaration in Example 2.9, a slightly modified version of the one used by the actual XML specification, tells us that the document type is `spec` and that the declarations are stored in a file called `spec.dtd`.

Example 2.9: Sample document type declaration

```
      <!DOCTYPE spec SYSTEM "spec.dtd">
```

See 2.8, “Prolog and Document Type Declaration”, for more on these.

- *Processing instructions* are special instructions for the application. See 2.6, “Processing Instructions” for more information.

The remainder of 2.4, “Character Data and Markup”, describes five general entities that are so important that you don’t have to declare them because they’re automatically predeclared for you: `&amp;`, `&lt;`, `&gt;`, `&apos;` and `&quot;`. See 4.6, “Predefined Entities”, for more on these.

The ampersand character (&) and the left angle bracket (<) may appear in their literal form *only* when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. They are also legal within the literal entity value of an internal entity declaration; see **Section 4.3.2: Well-Formed Parsed Entities**. If they are needed elsewhere, they must be escaped using either numeric character references or the strings “`&amp;`” and “`&lt;`” respectively. The right angle bracket (>) may be represented using the string “`&gt;`”, and must, for compatibility, be escaped using “`&gt;`” or a character reference when it appears in the string “`]]>`” in content, when that string is not marking the end of a CDATA section.

This lists five situations where you can use the ampersand and left angle bracket characters as they are (that is, the actual `"&"` and `"<"` characters instead of their entity references `&amp;` and `&lt;`):

- In the markup they were intended for: the ampersand as the beginning of an entity reference, and the left angle bracket as the beginning of a tag, comment, declaration, or processing instruction.
- Within a comment, where they won’t be treated as markup, but as their plain old selves. For example, the ampersand and less-than symbol in Example 2.10 won’t cause any problems.

Example 2.10: Ampersand and less-than symbol within a comment—no problem

```
        <!-- if a = 2 & b = 4, then a < b -->
```

- In a processing instruction, where they're no more trouble than in a comment.
- In a CDATA section, where they won't be treated as markup, because after all, that's the point of CDATA sections.
- Inside an internal entity's replacement text. For example, if an `ltref` entity is defined with the declaration shown in Example 2.11, the entity reference `&ltref;` in a document will actually be replaced by "&lt;" instead of by "<" because the ampersand in the entity declaration's literal entity value (the part between the quotes in the declaration) was treated as a data ampersand and not as the beginning of an entity reference. That's why they call it a "literal entity value": because most characters in it are treated literally.

Example 2.11: An ampersand in an internal entity's replacement text

```
<!ENTITY ltref "&lt;">
```

To include an ampersand or left angle bracket[†] as data in a document it must be "escaped". This term, which has been used in programming languages for years, describes a way of letting markup characters "escape" the parser so that it doesn't treat them as markup.

[†] A "left angle bracket" is also known as a "less-than" symbol, which inspires the `lt` abbreviation used in its entity reference, but the XML specification uses the term "left angle bracket" when referring to the "<" character. Similarly, it prefers the term "right angle bracket" to "greater-than character" for the ">" character, despite the `gt` abbreviation used for its predeclared entity reference.

For the ampersand and left angle bracket, you use the same trick that you would use to put the "ñ" in "España" or the "ä" in "bräu": use each character's entity reference. For `ñ`, it's `&ntilde;`, for `ä`, it's `&auml;`, for `&`, it's `&amp;`, and for `<`, it's `&lt;`.†

The same applies to the right angle bracket, or "greater-than" (">") character, although this is rarely necessary in a document's character data. A parser can't mistakenly treat it as the end of a tag, processing instruction, comment, or declaration because it isn't recognized unless there's an unfinished markup string in progress. You do need the greater-than symbol's entity reference (`&gt;`) after the characters "]]" if your document just happens to need a "]]>" somewhere that isn't ending a CDATA section. See 2.7, "CDATA Sections" for more on these.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, "`]]>`".

Now we have a more specific definition of character data, or rather, two definitions, depending on the context:

- Between an element's start- and end-tags (that is, in an element's content), any string where no markup begins. You can't have the end of markup (for example, a `>` character to represent the end of a tag) unless you recently began some markup, because the parser won't even try to treat `>` as the end of a tag unless it was looking for one.

† One nice feature of the `&amp;` and `&lt;` entity references is that they are among the five special ones that don't need to be declared unless you want to maintain interoperability with pre-WebSGML SGML systems. 4.6, "Predefined Entities", provides further background on this.

- In a CDATA section, every thing is character data except for the string `]]>`, which means "end of this CDATA section".

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "`&apos;`", and the double-quote character (") as "`&quot;`".

The ability to delimit strings with either single- or double-quotes usually means that if you need one of these in your string you delimit that string with the other. What if you need both in the string? Use either to delimit and use the described entity references within the string, as demonstrated in 2.12.

Example 2.12: Using the `&apos;` and `&quot;` entity references to insert quotes in an entity's replacement text

```
<!ENTITY inferno "Dante&apos;s &quot;Inferno&quot;">
```

`quot` and `apos` are predefined entities, so there is no need to declare them before referencing them unless you want to maintain interoperability with pre-WebSGML SGML systems.

Character Data

```
[14]CharData        ::=  [^<&]* - ([^<&]* ']]>' [^<&]*)
```

In the concise language of productions, production 14 tells us what the text preceding it already told us: that character data is any string of characters excluding the ampersand, the less-than symbol, and the "]]>" string.

2.5. Comments

Comments may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may, but need not, make it possible for an application to retrieve the text of comments. For compatibility, the string "`--`" (double-hyphen) must not occur within comments.

You can put them in document type declarations "at places allowed by the grammar"—but where does it allow them? Production 28 shows that markup declarations (**`markupdecl`** in the production) can be part of a document type declaration, and production 29 shows that comments are one type of markup declaration.

As with the comments in computer programs, the processor may just ignore their content. They're often there as notes from the authors to themselves or to others on their writing team, like the one shown in Example 2.13.

Example 2.13: Sample comment within a `par` element

```
<par>The W3C approved XML as an official Recommendation
on February 10, 1998.<!-- double-check that date -->
</par>
```

Comments

```
[15]Comment          ::=  '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
```

An example of a comment:

```
<!-- declarations for <head> & <body> -->
```

Note how this example demonstrates what 2.4, "Character Data and Markup" said about the < and & characters being allowed within comments.

2.6. Processing Instructions

Processing instructions (PIs) allow documents to contain instructions for applications.

The parser may or may not pass XML comments to the application, but it must pass processing instructions, because that's the purpose of processing instructions: to represent special instructions for the application.

Processing Instructions

[16]**PI**	::=	'<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17]**PITarget**	::=	Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))

PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (`PITarget`) used to identify the application to which the instruction is directed. The target names "`XML`", "`xml`", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets.

The processing instruction in Example 2.14 tells a mythical "stinker" application to generate a particular scent for five seconds.

Example 2.14: A sample processing instruction

```
<?stinker scent="newcar.sml" time="5 secs" ?>
```

A particular document may have processing instructions for several different applications, so the processing instruction target (**`PITarget`**), right after the opening <?, identifies the target application for this processing instruction.†

† The XML Working Group considered requiring notation declarations for each processing instruction target, but this would have effectively prohibited the use of processing instructions in well-formed documents, which don't require any declarations. See 4.7, "Notation Declarations", for more information

2.7. CDATA Sections

CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`":

As we saw in 2.4, "Character Data and Markup", to "escape" some text is to identify it as something that should escape parsing. In other words, if there's anything in that text that would normally be considered XML markup, treat it as character data. After an XML parser sees the `<![CDATA[` sequence that indicates the beginning of a CDATA section and before it sees the `]]>` markup that indicates the end, it assumes that all the characters it sees are character data—even any left angle brackets and ampersand characters.

CDATA Sections

[18]CDSect	::=	`CDStart CData CDEnd`
[19]CDStart	::=	`'<![CDATA['`
[20]CData	::=	`(Char* - (Char* ']]>' Char*))`
[21]CDEnd	::=	`']]>'`

Within a CDATA section, only the `CDEnd` string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "`&lt;`" and "`&amp;`". CDATA sections cannot nest.

CDATA sections are popular for showing demonstration XML (or SGML or HTML) markup within an XML document. The markup can be shown as-is with no modifications, but the parser won't confuse the demonstration markup with actual document markup.

For example, in the document fragment in Example 2.15, an XML parser won't consider `<center>`, `<img src="stoppages.jpg">`, or

`</center>` to be actual markup within the document's `par` element because that text is inside a CDATA section.

Example 2.15: Use of CDATA section to "escape" `img` and `center` tags

```
<par>This HTML code will center the "Standard Stoppages" picture:
<![CDATA[ <center>
<img src="stoppages.jpg">
</center>
]]> </par>
```

The spec tells us that, in CDATA sections, left angle brackets and ampersands "need not (and cannot) be escaped using '`&lt;`' and '`&amp;`'". The strings `&lt;` and `&amp;` are just character data in a CDATA section, just like everything else other than `]]>`.

"CDATA sections cannot nest" means that you can't put one CDATA section inside of another. If you think about how CDATA sections work, nesting them doesn't make any sense. For example, in Example 2.16, the CDATA start at line 2 means "this is all character data until the next `]]>`".

Example 2.16: Attempted nesting of CDATA sections

```
1.  This line isn't in any CDATA section.
2.  <![CDATA[
3.  This is inside of a CDATA section.
4.  <![CDATA[
5.  This is inside of a nested one, which is illegal in XML.
6.  ]]>
7.  This is outside of the inner nested CDATA section, but
8.  still inside the outer one.
9.  ]]>
10. This line isn't in any CDATA section.
```

The next `]]>` after line 2 is on line 6, so the attempt at starting a new CDATA section on line 4 is just treated by the processor as more character data. The real problem comes when the parser reaches line 9: if line 6 ended the CDATA section begun at line 2, what is line 9 ending? Nothing, so the parser doesn't know what to do—so it's an error.

One more handy thing about CDATA sections: a document with no document type declaration has no information about which elements have carriage returns that really matter. That's because it has no element type declarations to reveal whether an element type has element content (see 3.2.1, "Element Content", for more on this). CDATA sections become even more useful for these documents, because they clearly indicate which parts of a document should have their carriage returns and other white space left alone by the parser.

An example of a CDATA section, in which "`<greeting>`" and "`</greeting>`" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

Tip *Production 19 is the first production in the XML specification that uses an XML keyword: "CDATA". Note that it's written in upper-case in the production, with no option of writing it in lower-case. This applies to all XML keywords—they have to be written in upper-case.*

2.8. Prolog and Document Type Declaration

XML documents may, and should, begin with an *XML declaration* which specifies the version of XML being used. For example, the following is a complete XML document, well-formed but not valid:

```
<?xml version="1.0"?>
<greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The second “Hello, world” example’s status as a legal XML document demonstrates the potential simplicity (and much of the appeal) of XML. It’s well-formed “but not valid” because a valid document’s elements all conform to element type declarations in the document’s DTD, and this document doesn’t even have an associated DTD. See the upcoming specification paragraph beginning “The function of the markup in an XML document…” for more on this.

This section of the XML specification describes markup that can make a document even more useful, because it provides extra information to a processing program about the document and its structure. The first “hello world” example above shows the first thing that you “may, and should” add: an XML declaration, which tells the processor “Hey! This is an XML document! It conforms to version 1.0 of the XML specification!”

> The version number "`1.0`" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "`1.0`" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "`1.0`", but this intent does not indicate a commitment to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

In case readers are tempted to assume that the next generation of XML documents will begin with `<?xml version="1.1">` or `<?xml version="2.0">`, this rather legalistic paragraph warns them not to make any such assumptions. In fact, it tells them not to assume that there will even *be* another version of XML.

Imagine that you wrote an XML processing application, and now it's three years later and people are still using that program. Perhaps the XML spec has been updated to version 1.2, and these users have some new documents that take advantage of version 1.2's new features. What if they feed the new documents to your program that they have grown to love over the years? How does your program react to the use of these new features? There are two choices:

- It may sputter and choke on the new parts of the document.
- It can first check for the XML declaration's version number and output a warning message if it equals anything other than "1.0".

> The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units. An XML document is *valid* if it has an associated document type declaration and if the document complies with the constraints expressed in it.

This is probably the most important paragraph in the whole specification. Markup identifies storage structure (entity structure) and logical structure (a document's elements and their relationship) and specifies the attribute values that go with each element. By doing this, it makes it easier for software to manipulate a document for different purposes, thereby making the document a more valuable asset.

Software can do even more with documents if it knows their structure—that is, which elements are made up of which other elements and the ordering of the component elements. A document type declaration tells the processing program the definition of a document's structure, or, as database people say, the "schema". By "logical constraints", the specification refers to the definition of an element type's makeup, such as "a chapter element is made of a title element followed by one or more section elements". This is a constraint because in a well-formed XML document that isn't valid, like the first "hello world" example above, you could put any elements you like between a `<chapter>` start-tag and a `</chapter>` end-tag. Defined constraints help a processing program know what to expect.

To "support the use of predefined storage units" is to provide a way to reference files and other storage units from within a document. A document can have many reasons for identifying an external file:

- It may be additional marked-up XML content.
- It may store a stylesheet for the document.
- It may store a picture, sound, or audio file that should be made available to anyone viewing the document.

A document type declaration allows this by making it possible to declare entities for use within a document.

A valid XML document declares a document type and conforms to the logical and physical structure (that is, element and entity structure) defined for that document type. For example, the document in Example 2.17 declares a document type of `rpt`. The document instance comes after the document type declaration, which contains the various element and entity declarations.

Example 2.17: Document with internal declaration subset

```
<?xml version="1.0"?>
<!DOCTYPE rpt [
<!ELEMENT rpt   (title,par+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT par   (#PCDATA)>
<!ENTITY  auml "[auml  ]">
<!ENTITY  disclaimer SYSTEM "disclaimer.xml">
<!ENTITY  copyright  SYSTEM "copyright.xml">
]>
<rpt><title>Snee: A White Paper</title>
&copyright;
<par>Here is the first paragraph. The German word
for "brew" is "br&auml;u."</par>
<par>Here is the second paragraph.</par>
&disclaimer;
</rpt>
```

The document type declaration must appear before the first element in the document.

In Example 2.17, the document's first element (`rpt`) doesn't start until after the end of the document type declaration.

Prolog

[22]**prolog**	::=	XMLDecl? Misc* (doctypedecl Misc*)?
[23]**XMLDecl**	::=	’<?xml’ VersionInfo EncodingDecl? SDDecl? S? ’?>’
[24]**VersionInfo**	::=	S ’version’ Eq (’ VersionNum ’ | " VersionNum ")
[25]**Eq**	::=	S? ’=’ S?
[26]**VersionNum**	::=	([a-zA-Z0-9_.:] | ’-’)+
[27]**Misc**	::=	Comment | PI | S

The prolog can provide advance knowledge of the document: the version of XML being used and the structure of the document. It can, but doesn’t have to tell us either of these, because as production 22 shows us, the XML declaration and the DOCTYPE declaration are both optional. A prolog is better off including both, because the XML declaration and DOCTYPE declaration both provide valuable information to the XML processor.

In the document in Example 2.18, everything except line 5 is the prolog. In terms of production 22, the first line has the **`XMLDecl`** followed by a comment, which according to production 27 qualifies as a **`Misc`**. Next comes a **`doctypedecl`** at lines 2 through 4, and then another **`Misc`** after the `]>` that closes the **`doctypedecl`** before the brief document element on line 5. Although you can’t see anything after line 4’s `]>`, look again at production 27 for **`Misc`**: the third and last choice is **`S`**, for white space, and the carriage return at the end of line 4 counts.

Example 2.18: XML document with four-line prolog

```
1. <?xml version="1.0"?><!-- sample XML document -->
2. <!DOCTYPE verse [
3. <!ELEMENT verse (#PCDATA)>
4. ]>
5. <verse>She smoothes her hair with automatic hand</verse>
```

Because single or double quotation marks in a production enclose text that must be included literally, production 24 above tells you that Example 2.19 would be a legal **`VersionInfo`**, but it wouldn’t. The production's author meant (but failed) to show that single or double quotes had to be included around the version number. The two correct versions are shown in Example 2.20.

Example 2.19: Incorrect `VersionInfo` markup

```
version=1.0
```

Example 2.20: Two possible correct versions of `VersionInfo` markup

```
version="1.0"
version='1.0'
```

To show this, the production should have done something like the revision shown in 2.21, essentially quoting the quotes to show that they should be literally included.[†]

Example 2.21: Revised version of production 24

```
VersionInfo ::= S 'version' Eq
                ("'" VersionNum "'" | '"' VersionNum '"')
```

> The XML *document type declaration* contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or *DTD*. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.

In Example 2.17, the declaration "contains or points to markup declarations"—it *contains* the markup declarations between the square braces (`[]`). In 2.22, it *points to* the `rpt.dtd` file that has the necessary declarations. The DTD file is shown in Example 2.23.

[†] The spec's authors are aware of the error.

Example 2.22: XML document with an external declaration subset

```
<?xml version="1.0"?>
<!DOCTYPE rpt SYSTEM "rpt.dtd">
<rpt><title>Snee: A White Paper</title>
&copyright;
<par>Here is the first paragraph. The German word
for "brew" is "br&auml;u."</par>
<par>Here is the second paragraph.</par>
&disclaimer;
</rpt>
```

Example 2.23: `rpt.dtd` file referenced in Example 2.22

```
<!ELEMENT rpt (title,par+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT par (#PCDATA)>
<!ENTITY auml "[auml  ]">
<!ENTITY disclaimer SYSTEM "disclaimer.xml">
<!ENTITY copyright  SYSTEM "copyright.xml">
```

These declarations provide a "grammar": a body of rules about the allowable ordering of this document's "vocabulary" of element types. We call this grammar a DTD, or "Document Type Definition". It is also common to refer to the declarations that define the grammar as the DTD, which sometimes causes confusion.

When the `rpt` DTD is in a separate `rpt.dtd` file from the "Snee: A White Paper" document, that DTD is an external entity just like the `disclaimer.xml` and `copyright.xml` files that the `disclaimer` and `copyright` declarations point to. (Compare this with `auml`, which is an internal entity because its value of `[auml  ]` is right in there with the declarations and not in a separate file.)

The `rpt.dtd` DTD file is described as a "special kind" of external entity because unlike `disclaimer.xml` or `copyright.xml` it is an external entity that holds a subset of a document type declaration. In Example 2.17, all the document type declaration's markup declarations were in an internal subset—that is, they were part of the document file (or rather, part of the document entity) itself.

As the specification tells us, a single document type declaration can both contain an internal subset and point to an external subset. For example, if the `rpt.dtd` file only consisted of the three lines shown in Example 2.24, the document shown in Example 2.25 would still be fine. This is because it both points to the smaller `rpt.dtd` external subset and also has the remaining declarations in its own internal subset.

Example 2.24: Sample three-line `rpt.dtd` file

```
<!ELEMENT rpt (title,par+)>
<!ELEMENT par (#PCDATA)>
<!ENTITY auml "[auml  ]">
```

Example 2.25: Document referencing 2.24 along with its own internal declaration subset

```
<?xml version="1.0"?>
<!DOCTYPE rpt SYSTEM "rpt.dtd" [
<!ELEMENT title (#PCDATA)>
<!ENTITY disclaimer SYSTEM "disclaimer.xml">
<!ENTITY copyright SYSTEM "copyright.xml">
]>
<rpt><title>Snee: A White Paper</title>
&copyright;
<par>Here is the first paragraph. The German word
for "brew" is "br&auml;u."</par>
<par>Here is the second paragraph.</par>
&disclaimer;
</rpt>
```

In a case like this, the last sentence of the specification paragraph above tells us that the “DTD for a document consists of both subsets taken together”. More precisely, the DTD is declared by the markup declarations in both subsets taken together.

> A *markup declaration* is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. These declarations may be contained in whole or in part within parameter entities, as described in the well-formedness and validity constraints below. For fuller information, see **Section 4: Physical Structures**.

The document in Example 2.26 demonstrates all the categories of markup declaration. Line 10's element type declaration shows that a `tale` element (the root element, as shown by the DOCTYPE declaration on line 2) is made of a `title` element followed by one or more `par` and `illus` elements.

The `par` element's declaration is actually a string stored in the `pardecl` entity by the declaration at line 8; the `%pardecl;` entity reference at line 12 has the effect of putting that `par` element type declaration right there between the `title` element type declaration on line 11 and the `illus` element type declaration on line 13. This `pardecl` declaration is an example of a parameter entity, because unlike a general entity that stores a piece of a document instance, a parameter entity stores a piece of a document type declaration—in this case, the `par` element type declaration.

Example 2.26: Sample document with all categories of markup declaration

```
 1. <?xml version="1.0"?>
 2. <!DOCTYPE tale [
 3.
 4. <!NOTATION EPS PUBLIC "+//ISBN 0-201-18127-4::Adobe//
 5. NOTATION PostScript Language Ref. Manual//EN">
 6.
 7. <!ENTITY glow SYSTEM "img/glow.eps" NDATA EPS>
 8. <!ENTITY % pardecl "<!ELEMENT par   (#PCDATA)>">
 9.
10. <!ELEMENT tale (title,(par|illus)+)>
11. <!ELEMENT title (#PCDATA)>
12. %pardecl;
13. <!ELEMENT illus EMPTY>
14. <!ATTLIST illus picfile ENTITY #REQUIRED>
15. <!-- End of document type declaration -->
16. ]>
17.
18. <tale><title>What the Thunder Said</title>
19. <par>After the torchlight red on sweaty faces</par>
20. <illus picfile="glow"/>
21. </tale>
```

The other entity declaration, on line 7, declares the file `glow.eps` in the `img` subdirectory as an entity to be used as needed in the document. The `illus` element type's attribute-list (ATTLIST) declaration on line 14 declares one attribute for the `illus` elements: `picfile`, an attribute whose attribute type of ENTITY shows that each `illus` element must have a declared entity name as its value.

The document's one `illus` element, on line 20, has a `picfile` value of `glow`, which was the first entity to be declared in this document's document type declaration at line 7. This `glow` entity declaration shows that it's an EPS file, but what's an EPS file? This brings us to another category of markup declarations: notation declarations, which identify the format of "unparsed" data (that is, data that the processor shouldn't parse as part of the XML text of this document). Example 2.26's notation declaration on lines 4 and 5 tells us where to find the details on the PostScript format.

Chapter 4, "Physical Structures" and 4.1, "Character and Entity References" describe the use of parameter entities; for more background on the other kinds of declarations, see 3.2, "Element Type Declarations"; 3.3, "Attribute-List Declarations"; 4.2, "Entity Declarations"; and 4.7, "Notation Declarations".

Document Type Definition

[28]doctypedecl	::=	'<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | PEReference | S)* ']' S?)? '>'	[VC: Root Element Type]
[29]markupdecl	::=	elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment	[VC: Proper Declaration/PE Nesting] [WFC: PEs in Internal Subset]

We've seen examples of all the **`doctypedecl`** components:

- **`S`**, according to production 3, is made up of one or more space characters.
- In Example 2.25 earlier, `SYSTEM "rpt.dtd"` is an example of an `ExternalID`.
- Each line between Example 2.25's square brackets ("`[]`") is a markup declaration, or **`markupdecl`**.
- A **`PEReference`** is a parameter-entity reference, like `%pardecl;` in Example 2.26.

The **`markupdecl`** production shows that it may be one of six things: one of the four declaration types demonstrated by the `tale` example, a comment, or a processing instruction (**`PI`**). Comments (described further in 2.5, "Comments") have no information for the parser, and usually give background to a person reading an XML document. In Example 2.26's `tale` document, "End of document type declaration" on line 15 is a comment.

Processing instructions contain data that the parser passes on to the system, to the processing application, or to both. The very first line of the `tale` document, which identifies the release of XML being used, is a processing instruction. See 2.6, "Processing Instructions" for more background.

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (`elementdecl`, `AttlistDecl`, and so on) describe the declarations *after* all the parameter entities have been included.

A parameter entity's replacement text is the resulting text after all applicable replacements have been made (see 4.5, "Construction of Internal Entity Replacement Text", for more on this). In Example 2.26's `tale` example, the `parmodel` parameter entity's replacement text is `<!ELEMENT par (#PCDATA)>`. (See 3.2.2, "Mixed Content", for more on "PCDATA".)

"Nonterminals" are the pieces of a document whose structure is shown by the specification's productions; see "" for more on productions and nonterminals.

The final sentence of the paragraph above tells us that the upcoming productions in the XML specification do not show you where you might put parameter entity references. Instead, they assume that any parameter entity references that may have been in the nonterminal shown have had their replacement text substituted for them.

VALIDITY CONSTRAINT: Root Element Type

The `Name` in the document type declaration must match the element type of the root element.

This is the first of many validity constraints in the XML specification. Productions refer to these constraints (production 28 cites this one) to show further conditions that the defined nonterminal must meet for a parser to consider its document valid. (Well-formedness constraints play a similar role in defining well-formedness.)

This validity constraint tells us that the **`Name`** in a document type declaration (production 22) can't be just any Name (2.3, "Common Syntactic Constructs", shows exactly what the specification means by

“Name”). It has to be the element type name of the root element of the document: that is, the single main element that encloses all of the document’s other elements.

VALIDITY CONSTRAINT: Proper Declaration/PE Nesting

Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (`markupdecl` above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text.

If a parameter entity has the beginning or end of a markup declaration, it has to have the other one as well.

We’ll see in the next well-formedness constraint that there are certain conditions for storing and using a piece of a markup declaration instead of an entire one. But even when meeting those conditions, the piece being stored can never have the beginning of a markup declaration without also including that declaration’s ending, or vice versa. (We’ll also see examples after the next well-formedness constraint.)

WELL-FORMEDNESS CONSTRAINT: PEs in Internal Subset

In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)

When you use a parameter entity reference in a document type declaration’s internal subset, it can’t represent merely a portion of a markup declaration. It can when used in an external subset.

This is easier to see with an example. The `ents.xml` file shown in Example 2.27 has an internal subset and refers to the external subset stored in the file `ents-ext.dtd` shown in Example 2.28. (Both files have the interesting parts described in comments and the illegal parts commented out so that the examples will parse properly; more detailed descriptions follow `ents-ext.dtd`.)

Example 2.27: Parameter entities in internal and external subsets

```
<?xml version="1.0"?>
<!DOCTYPE a SYSTEM "ents-ext.dtd" [
<!ELEMENT a (#PCDATA|b|c|d|e|f|g)*>

<!-- The following two lines work fine. -->
<!ENTITY % edecl "<!ELEMENT e (#PCDATA)>">
%edecl;

<!-- The following two declarations are illegal,
     so they’re commented out and replaced by the
     line after them. Note that ents-ext.dtd’s
     equivalent of this, cdeclpart, works fine.
<!ENTITY % fdeclpart "f (#PCDATA)">
<!ELEMENT %fdeclpart;>
-->
<!ELEMENT f (#PCDATA)>

<!ELEMENT g (#PCDATA)>
]>
<a>
<b>How</b> <c>about</c> <d>those</d>
<e>parameter</e> <f>entity</f> <g>rules.</g>
</a>
```

Example 2.28: The `ents-ext.dtd` file referenced in Example 2.28

```
<!ENTITY % bdecl "<!ELEMENT b (#PCDATA)>">
%bdecl;

<!-- The following two lines work fine, because
     they’re in an external declaration subset. -->
<!ENTITY % cdeclpart "c (#PCDATA)">
<!ELEMENT %cdeclpart;>

<!-- The following two lines are illegal, so they’re
     commented out and replaced by the line after them.
<!ENTITY % ddeclpart "<!ELEMENT d ">
```

```
12. %ddeclpart; (#PCDATA)>
13. -->
14. <!ELEMENT d (#PCDATA)>
```

The first entity declaration in Examples 2.27 and 2.28 each store an entire element type declaration in a parameter entity: `edecl` at line 6 of `ents.xml` and `bdecl` at line 1 of `ents-ext.dtd`. The lines immediately following each of these have references to these entities, in effect declaring the `e` and `b` element types whose declarations they store.

Both files then try to declare and use a parameter entity that stores several parameters of an element type declaration. `ents-ext.dtd` declares `cdeclpart` as `c (#PCDATA)` at line 6 and uses it on the following line, and an XML parser has no problem with this. It would have a problem with the `ents.xml` file's declaration and usage of `fdeclpart` at lines 13 and 14 because of the last paragraph of the spec shown above: it's referenced within a markup declaration in an internal subset. The `cdeclpart` parameter entity was not a problem because it was referenced in an external subset.

The final declaration doesn't work in the external subset, so the internal subset doesn't even try it: the `ents-ext.dtd` file's `ddeclpart` entity tried to store the beginning and a middle piece of the `d` element type's declaration at line 11. It didn't work because it was declared illegal by the "Proper Declaration/PE Nesting" validity constraint. (Because it didn't work, I commented it out.)

As the parenthesized final sentence of the specification paragraph above tells us, the "PEs in Internal Subset" well-formedness constraint doesn't apply to parameter-entity references in an external parameter entity or in a DTD external subset, because when an XML processor is checking for well-formedness, it doesn't have to bother with external parameter entities and DTD external subsets.

> Like the internal subset, the external subset and any external parameter entities referred to in the DTD must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol `markupdecl`, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed in the internal subset.

The first sentence here sums up what we saw in the Examples 2.27 and 2.28. The second sentence tells us that conditional sections are OK in an external subset but not in an internal subset. Conditional sections (described further in 3.4, "Conditional Sections") let you easily change whether a parser ignores or parses a large block of text. For example, changing the "INCLUDE" to "IGNORE" in Example 2.29 tells the parser not to parse anything between the second `[` and the `]]>` that shows where the conditional section ends.

Example 2.29: An INCLUDE marked section

```
<![INCLUDE[
<!ENTITY sss "Shantih shantih shantih">
<!ENTITY tsepic SYSTEM "img/tse.eps" NDATA EPS>
]]>
```

External Subset

[30]extSubset	::=	`TextDecl? extSubsetDecl`
[31]extSubsetDecl	::=	`( markupdecl | conditionalSect | PEReference | S )*`

Production 30 shows that an external subset consists of an optional text declaration followed by an external subset declaration. Production 31 shows that the latter is a combination of zero or more of the markup declarations, conditional sections, and parameter-entity references (with optional white space between them) that we've seen throughout 2.8, "Prolog and Document Type Declaration".

As explained in 4.3.1, "The Text Declaration", a text declaration is a processing instruction telling us the version of XML being used by a particular external parsed entity.

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

This restates something that is implied by the "PEs in Internal Subset" well-formedness constraint and demonstrated by the parameter entities in Examples 2.27 and 2.28: references to parameter entities that contain incomplete pieces of markup declarations are only legal in external subsets.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
<greeting>Hello, world!</greeting>
```

The system identifier "`hello.dtd`" gives the URI of a DTD for the document.

As we saw in the "Status of this Document" section at the beginning of the specification, a "Uniform Resource Identifier" (URI) is a notation for naming resources on the Web. A "Uniform Resource Locator" (URL) such as `http://www.w3.org` is one kind of URI. Even a simple file name can be treated as a Relative URL that points to a file in the same directory as the entity in which the Relative URL is stored. URLs are a good way to identify system identifiers because of the huge HTML world's familiarity with them and because of the ease with which a URL can point to both local and remote addresses.

So, it looks like all those file names we've been seeing in DOCTYPE declarations to identify the document type declarations' external subsets (like `hello.dtd` in the sample above) were URIs all the time.

Knowing this opens up a lot of possibilities. It means that your document can point to a DTD somewhere else by using a URL, like the one in Example 2.30.

Example 2.30: DOCTYPE declaration using a URL to identify a DTD

```
<!DOCTYPE min
    SYSTEM "http://www.snee.com/dtds/invoice.dtd">
```

This will be common for geographically widespread organizations that want to retain a regular structure for their documents. Different branch offices can point to a centrally maintained collection of DTDs without needing to maintain and update their own copies.

An XML processor treats a relative URI as being relative to the entity where it's stored, not relative to the document entity ultimately containing the reference. For example, say the `catalog.xml` document entity in the `grandfather` directory references the `parts.xml` file in the `father` subdirectory of `grandfather` as `father/parts.xml`, and an entity declaration in `parts.xml` references a file in `son/bolts.xml`. The XML processor will expect `son` to be a child of the `father` directory and not a child of the `catalog.xml` document's `grandfather` directory.

The declarations can also be given locally, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

The internal subset pre-empts any attempts to re-declare the same objects. In addition, entities declared in the internal subset can be referenced in the external subset.

Using the `extdecl.dtd` file shown in Example 2.31 as an external subset, this precedence of internal declarations will cause an XML processor to treat the document in Example 2.32 as if the `rutle` entity had the value “Barry” and the single `redecl` element’s `flavor` attribute had the default value of “lemon”.

Example 2.31: `extdecl.dtd` file with declarations pre-empted by Example 2.32’s internal subset declarations

```
<!ATTLIST redecl flavor CDATA "mint">
<!ENTITY  rutle "Stig">
```

Example 2.32: Document with internal subset declarations pre-empting external subset declarations

```
<?xml version="1.0"?>
<!DOCTYPE redecl SYSTEM "extdecl.dtd" [
<!ELEMENT redecl (#PCDATA)>
<!ATTLIST redecl flavor CDATA "lemon">
<!ENTITY rutle "Barry">
]>
<redecl>My favorite Rutle was &rutle;.</redecl>
```

2.9. Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations which appear external to the document entity.

An XML document can get by with no declarations at all. It can also have declarations as part of an internal subset, and it can have declarations in an external subset such as a separate DTD file. The standalone document declaration, or **`SDDecl`** (used by production 23, among others) answers the question “can we get by using this document without paying attention to the external declarations?”

Tip *Note that this question may not even be asked—the* **`SDDecl`** *provides information in case it’s requested; it doesn’t mandate behavior.*

Standalone Document Declaration

[32]`SDDecl`	`::=`	`S ’standalone’ Eq (("’" (’yes’ | ’no’) "’") | (’"’ (’yes’ | ’no’) ’"’))`	[VC: Standalone Document Declaration]

In a standalone document declaration, the value “yes” indicates that there are no markup declarations external to the document entity (either in the DTD external subset, or in an external parameter entity referenced from the internal subset) which affect the information passed from the XML processor to the application. The value “no” indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external *declarations*; the presence, in a document, of references to external *entities*, when those entities are internally declared, does not change its standalone status.

For example, the document type declaration in Example 2.33 tells us that, although the document may have declarations in an external file, the document can be processed without them. What kind of declarations are optional to document processing? The “Standalone Document Declaration” validity constraint below lists the possible conditions.

Example 2.33: XML declaration with a standalone declaration

```
<?xml version="1.0" standalone="yes"?>
```

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value “no” is assumed.

An external declaration subset that exists but isn’t necessary is really the exceptional case, which is why you can normally omit the standalone document declaration. If there is an external subset and your document needs it, the processor assumes that `standalone="no"` if you don’t specify otherwise; if there is no external subset, the standalone value is moot.

Any XML document for which `standalone="no"` holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

By “converted algorithmically”, this means that, with no human intervention, a document that depends on external declarations can be mechanically converted to one that doesn’t. Basically, such a program would just make a copy of the document with all the external declarations moved to the internal subset (all the external declarations, that is, not pre-empted in the internal subset). Doing so is likely to be common, because storing documents as multiple interrelated pieces often makes sense from a document management viewpoint, while delivering them from one computer to another (“network delivery applications”) is easier using documents that are self-contained units. These seemingly contradictory goals can both be achieved on a network by using document servers that can convert non-standalone documents to ones that don't need an external declaration subset.

VALIDITY CONSTRAINT: Standalone Document Declaration

The standalone document declaration must have the value "no" if any external markup declarations contain declarations of:

Now we get specific. Here are the four conditions in which a processor needs access to the markup declarations badly enough that if the declarations are in an external subset, the document being processed can't be considered standalone.

- attributes with default values, if elements to which these attributes apply appear in the document without specifications of values for these attributes, or

If the `chapter` element type has the attribute list definition shown in 2.34, then a `chapter` element in a document using this declaration has a `flavor` value of "mint" if its start-tag doesn't list any attribute values (`<chapter>`). A parser knows this from looking at the `chapter` element type's attribute list declaration, which specifies "mint" as the default value. If the parser has to look in an external declaration subset to find this declaration, then it's not a standalone document entity.

Example 2.34: Attribute declaration with default value of "mint"

```
<!ATTLIST chapter flavor CDATA "mint">
```

- entities (other than amp, `lt`, `gt`, apos, quot), if references to those entities appear in the document, or

If an XML processor finds the entity reference `&cnote;` in a document, how does it know what it refers to? It does so by looking at the `cnote` entity's declaration. If the processor has to look outside the document entity in an external declaration subset to find this declaration, then it's not a standalone document. Exceptions are the entity references used to represent the ampersand, less-than, greater-than, apostrophe, and quotation characters, because an XML processor will already know what these refer to. See 4.6, "Predefined Entities", for more on these.

- attributes with values subject to *normalization*, where the attribute appears in the document with a value which will change as a result of normalization, or

Some attribute values can refer to entities, and an important job of attribute value normalization is the resolution of these references. (See 3.3.3, "Attribute-Value Normalization", for more on this.) As with most other entity references, an XML processor needs their declarations to find them in storage. Any need to look at an external declaration subset for these means that the document is not a standalone one.

- element types with element content, if white space occurs directly within any instance of those types.

An element type consisting of element content has only other elements as children, with no character data that is not part of any child element (see 3.2.1, "Element Content", for background on this). Because the processor needs access to the element type declaration to know whether it should treat an element as having only element content, any need to look to an external declaration subset for the declaration means that the document is not a standalone one.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

2.10. White Space Handling

In editing XML documents, it is often convenient to use “white space” (spaces, tabs, and blank lines, denoted by the nonterminal S in this specification) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, “significant” white space that should be preserved in the delivered version is common, for example in poetry and source code.

For example, production 28 has many places where **`S`** indicates required white space. Because **`S`** means “one or more of the space, tab, and line end characters”, the parser doesn’t care about the difference between the DOCTYPE declaration shown in Example 2.35 and the one shown in Example 2.36.

Example 2.35: DOCTYPE declaration with minimum required spaces

```
<!DOCTYPE rant SYSTEM "rant.dtd">
```

Example 2.36: DOCTYPE declaration with extra spaces in it

```
<!DOCTYPE     rant
         SYSTEM     "rant.dtd"  >
```

To “set apart the markup for greater readability” refers to the practice of adding disposable spaces to make the marked-up document easier to read. If a discography document type’s `album` element type is declared as shown in Example 2.37 then `album` has element content (see 3.2.1, "Element Content", for more on this). A parser would not care whether an `album` element looks like the one in Example 2.38 or the one in Example 2.39.

Example 2.37: Declaration for `album` element shown in Examples 2.38 and 2.39

```
<!ELEMENT album (song+)>
```

Example 2.38: `album` element conforming to Example 2.37's element type declaration

```
<album><song>Hold My Hand</song><song>Number One</song>
<song>Love Life</song><song>Cheese and Onions</song></album>
```

Example 2.39: Another `album` element conforming to Example 2.37's element type declaration

```
<album>
   <song>Hold My Hand</song>
   <song>Number One</song>
   <song>Love Life</song>
   <song>Cheese and Onions</song>
</album>
```

The carriage returns and indentation in Example 2.39 don't matter to the XML processor.

Sometimes carriage returns and extra spaces are important, and you don't want the processor to throw them out. As examples, the specification mentions poetry (see Example 2.40) and the source code of programming languages. Consider the little Perl program shown in Example 2.41.

Example 2.40: Poem excerpt with meaningful white space

```
<poem>
<verse>'What shall we ever do?'</verse>
<verse>                 The hot water at ten.</verse>
<verse>And if it rains, a closed car at four.</verse>
</poem>
```

Example 2.41: Perl program with meaningful white space

```
#!/usr/local/bin/perl
$i = 0;
while (<>) {
   $i++;
   print "$i:     $_\n";
}
```

White space can be particularly important in program source code. While the difference between one and three spaces after the keyword "DOCTYPE" in a DOCTYPE declaration may not matter, such a cavalier attitude toward the space beginning some program listing lines would make the program difficult to read, and doing it to the space between the quotation marks in the program in Example 2.41's `print` line would change how the program worked.

The XML Working Group debated long and hard about how XML processors should handle white space characters, especially carriage returns.† Issues such as completely blank lines and carriage returns before and after comments make it difficult to lay out simple rules about which white space to preserve and which to throw out. After all

† At one point in their e-mail discussion of white space and the Record Start/Record End "characters" that delimit input lines, XML specification co-editor Tim Bray wrote "At this point I'd rather write WordPerfect macros than read another 10 postings about RS/RE".

the debate, the Working Group came up with something simple and straightforward.

An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content.

Instead of deciding which white space to throw out and which to keep, an XML processor must pass it all to the application.

Earlier drafts of the specification assigned further responsibilities to a validating XML processor: in addition to telling the application which white space characters were in element content, it was supposed to "signal to the application that white space in element content is not significant". The deletion of this line removes the value judgment on element content white space (like the carriage returns after each `song` element in Example 2.39) while still requiring the processor to identify element content white space for the application to use or ignore as it wishes.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose only possible values are "`default`" and "`preserve`". For example:

```
<!ATTLIST poem   xml:space (default|preserve) 'preserve'>
```

The value "`default`" signals that applications' default white-space processing modes are acceptable for this element; the value "`preserve`" indicates the intent that applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overriden with another instance of the `xml:space` attribute.

This is a nice way to tell the application which elements should have their white space left alone. For most attributes that you define for elements, you'll end up specifying attribute values for each of the elements of that type. By specifying a default value of "`preserve`" in this attribute list declaration, the processor will treat every `<poem>` start-tag as if it said `<poem xml:space="preserve">` instead. You can override this default by starting a poem with the tag `<poem xml:space="default">`. (Don't confuse this concept of default attribute values with the permitted value of "`default`" for the `poem` element type's `xml:space` attribute.)

Speaking of overriding, the last sentence of the specification paragraph above tells us that an `xml:space` value applies to all of an element's child elements and their descendants unless you specify otherwise for a specific element. For example, the `xml:space` attribute declaration shown for the `poem` element type in the spec's example above tells an XML processor to keep the extra spaces in its child elements—for example, the `verse` elements of Example 2.42. That is, unless `verse` had been declared with an attribute specification overriding this `xml:space` setting as shown in Example 2.43.

Example 2.42: Poem excerpt whose `verse` space will be kept because of `poem` element types `xml:space`

```
<poem>
<verse>'What is that noise?'</verse>
<verse>              The wind under the door.</verse>
<verse>'What is that noise now? What is the wind doing?'</verse>
<verse>              Nothing again nothing.</verse>
```

Example 2.43: Specifying `xml:space` value for `verse` element type

```
 <!ATTLIST verse xml:space (default|preserve) 'default'>
```

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

If no such `xml:space` attribute was declared for the root element (the main document element that contains all the other elements in the document), you can't assume anything about what the processor will tell the application regarding the handling of spaces in that document.

Don't worry too much about `xml:space`, because an XML document that will be formatted for display on a page or screen will probably have a corresponding stylesheet. Part of the point of a stylesheet is to store far more sophisticated instructions about handling of white space than the `xml:space` attribute ever could. Besides, `xml:space` does not mandate any particular behavior, anyway; like the standalone document declaration, it merely passes a message along to be used if the application is interested.

2.11. End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

Most text processing programs treat a line of text as the basic unit of a text file. This is part of the legacy of punch cards, which represented each line of a file with a single card. (It's no coincidence that before computers used the windows, icons, and mouse pointers of graphical user interfaces, the old green text-mode computer screens showed up to 80 characters on each line—that's how many characters each punch card stored.)

Different operating systems represent the end of a line in different ways:

- UNIX machines use a line feed (byte 10, or in hexadecimal, "#xA").

- Macintoshes use a carriage return (byte 13, or "#xD" in hex).
- Windows PCs use a carriage return followed by a line feed.

This is why, when a program such as an FTP utility copies files from one computer to another, it often wants to know if they're text or binary—if the latter, they leave every byte alone, but for text files, they need to know about any necessary line end conversions.

In developing the XML specification, some members of the Working Group questioned whether the concept of a "line" was still relevant. After all, an XML document is a collection of elements, entities, and declarations. If a document's DTD has no `xml:space` attributes declared anywhere, a document without a single carriage return or line feed is functionally the same as a document with a line end character at the last word break before every eightieth character.

The Working Group decided to use the term "for editing convenience", because we still think of a document in terms of lines when we interact with it on the screen, or for that matter, on paper. (Think how often you use word processing commands that deal in terms of lines: jump to the current line's beginning, jump to its end, delete the current line, and so forth.)

To simplify the tasks of applications, wherever an external parsed entity or the literal entity value of an internal parsed entity contains either the literal two-character sequence "#xD#xA" or a standalone literal #xD, an XML processor must pass to the application the single character #xA. (This behavior can conveniently be produced by normalizing all line breaks to #xA on input, before parsing.)

No matter which representation of a line end is encountered by an XML processor, it passes along a single line feed character (ASCII character 10) to the application. Therefore, the XML application—unlike an FTP program—doesn't have to worry about different possible representations. This simplifying of the application's job is another example of the effort to ease the development of small yet effective applications.

Tip *The processor still understands all three representations of line ends, but it must always pass a single* #xA *(line feed) character to the application.*

2.12. Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be declared if it is used. The values of the attribute are language identifiers as defined by [IETF RFC 1766], "Tags for the Identification of Languages":

XML's country and language identifiers can give an application important information that it needs for tasks like case conversion, because two countries that speak the same language may have different case conversion rules. (For example, an upper-case "è" is "E" in Montreal but "È" in Paris.)

In eastern alphabets, knowing the specific language and country is particularly important, because certain Unicode characters may be used in Chinese, Japanese, or Korean language documents. Displaying them properly requires a knowledge of which language is in use.

Language Identification

[33]	**LanguageID**	::=	Langcode (’-’ Subcode)*
[34]	**Langcode**	::=	ISO639Code | IanaCode | UserCode
[35]	**ISO639Code**	::=	([a-z] | [A-Z]) ([a-z] | [A-Z])
[36]	**IanaCode**	::=	(’i’ | ’I’) ’-’ ([a-z] | [A-Z])+
[37]	**UserCode**	::=	(’x’ | ’X’) ’-’ ([a-z] | [A-Z])+
[38]	**Subcode**	::=	([a-z] | [A-Z])+

The Langcode may be any of the following:

- a two-letter language code as defined by [ISO 639], “Codes for the representation of names of languages”
- a language identifier registered with the Internet Assigned Numbers Authority [IANA]; these begin with the prefix “`i-`” (or “`I-`”)
- a language identifier assigned by the user, or agreed on between parties in private use; these must begin with the prefix “`x-`” or “`X-`” in order to ensure that they do not conflict with names later standardized or registered with IANA

The term “tag” in the title of the Internet Engineering Task Force’s (IETF) Request for Comment (RFC) 1766 has nothing to do with the XML sense of the term. This RFC defines a standard for identifying a language using one or more words: the first identifies the language and the optional second one identifies the country in which the language is being spoken and optional additional information. The language “tag” should be the two-letter code specified in ISO 639, “Codes for the representation of names of languages”. For example, “fr” is French, “en” is English, and “sa” is Sanskrit.

Tip *The* **`langcode`** *uses the two-letter language codes from version 1 of ISO 639, not version 2’s three-letter codes.*

> There may be any number of `Subcode` segments; if the first subcode segment exists and the Subcode consists of two letters, then it must be a country code from [ISO 3166], "Codes for the representation of names of countries". If the first subcode consists of more than two letters, it must be a subcode for the language in question registered with IANA, unless the `Langcode` begins with the prefix "`x-`" or "`X-`".

The optional second "tag" specifies the country using either an abbreviation from ISO 3166, "Codes for the representation of names of countries" (for example, "BE" for Belgium or "US" for the United States) or some other code registered with the Internet Assigned Numbers Authority (the group responsible for first-level domain names like `com`, `edu`, and `org`). According to IETF RFC 1766, subsequent "tags" can be anything you like.

> It is customary to give the language code in lower case, and the country code (if any) in upper case. Note that these values, unlike other names in XML documents, are case insensitive.

For example, if Celine Dion, Jean-Claude Van Damme, and Johnny Halliday were each going to author XML essays on Proust's use of smell imagery, the French-Canadian ballad belter would use a language code of `fr-CA` to indicate "Canadian French", the muscles from Brussels would use `fr-BE` to show that his essay was in Belgian French, and aging rock star Johnny Halliday, being the most French of the three, would use `fr-FR`. On the other hand, an XML document about the "Code Talkers" (the Navajo U.S. Marines who transmitted coded radio messages in the Pacific during World War II) could use the language code `i-navajo` because the Navajo language has an entry registered with the IANA. Or, if you and a client had agreed to transmit documents in pig latin, and found no existing ISO or IANA code for pig latin, you could make up and use your own, as long as you preceded it with `x-` or `X-` (for example, `x-pgl`).

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
  </sp>
```

The intent declared with `xml:lang` is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of `xml:lang` on another element within that content.

In the specification's example above, the four `l` elements written in German are children of the `sp` element. Because they have no `xml:lang` attribute of their own to specify their language, an application must treat them as if they had the `xml:lang` value of `de`, as their parent does.

A simple declaration for `xml:lang` might take the form

```
xml:lang  NMTOKEN  #IMPLIED
```

but specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the xml:lang attribute might be declared this way:

```
    <!ATTLIST poem   xml:lang NMTOKEN 'fr'>
    <!ATTLIST gloss  xml:lang NMTOKEN 'en'>
    <!ATTLIST note   xml:lang NMTOKEN 'en'>
```

Note that the attribute declared default of `#IMPLIED` in that first example makes that `xml:lang` attribute optional. For the `poem`, `gloss`, and `note` examples, including this attribute in element start-tags is also optional, but for a different reason: because defaults are supplied. If no `xml:lang` value is specified for any `poem`, `gloss`, or `note` elements in the document, they'll each have the default value shown in their declarations.