1 Syntax

Syntax is the lowest level of XML. To a large extent the contents of this section focus on the microstructure of XML. By investigating different ways to represent the same structures, the items answer the question, how can XML best be authored for maximum interoperability and convenience? Little of what's discussed here affects the information content of an XML document.

However, just as there are better and worse ways to write C code that produces identical machine code when compiled, so too are there better and worse ways to write XML documents. In this section we explore some of the most useful ways to improve the legibility, maintainability, and extensibility of your XML documents and applications.

Item 1 Include an XML Declaration

Although XML declarations are optional, every XML document should have one. An XML declaration helps both human users and automated software identify the document as XML. It identifies the version of XML in use, specifies the character encoding, and can even help optimize the parsing. Most importantly, it's a crucial clue that what you're reading is in fact an XML document in environments where file type information is unavailable or unreliable.

The following are all legal XML declarations.

```
<?xml version="1.0"?>
<?xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<?xml version="1.0" standalone="yes"?>
```

In general the XML declaration must be the first thing in the XML document. It cannot be preceded by any comments, processing instruction, or

even white space. The only thing that may sometimes precede it is the optional byte order mark.

The XML declaration is not a processing instruction, even though it looks like one. If you're processing an XML document through APIs like SAX, DOM, or JDOM, the methods invoked to read and write the XML declaration will not be the same methods invoked to read and write processing instructions. In many cases, including SAX2, DOM2, XOM, and JDOM, the information from the XML declaration may not be available at all. The parser will use it to determine how to read a document, but it will not report it to the client application.

Each XML declaration has up to three attributes.

1. `version`: the version of XML in use. Currently this always has the value 1.0, though there may be an XML 1.1 in the future. (See Item 3.)
2. `encoding`: the character set in which the document is written.
3. `standalone`: whether or not the external DTD subset makes important contributions to the document's infoset.

Like other attributes, these may be enclosed in single or double quotes, and any amount of white space may separate them from each other. Unlike other attributes, order matters. The `version` attribute must always come before the `encoding` attribute, which must always come before the `standalone` declaration. The `version` attribute is required. The `encoding` attribute and `standalone` declaration are optional.

The `version` Info

The `version` attribute always has the value 1.0. If XML 1.1 is released in the future (and it may not be), this will probably also be allowed to have the value 1.1. Regardless, you should always use XML 1.0, never version 1.1. XML 1.0 is more compatible and more robust, and it offers all the features XML 1.1 does. Item 3 discusses this in more detail.

The `encoding` Declaration

The `encoding` attribute specifies which character set and encoding the document is written in. Sometimes this identifies an encoding of the Unicode character set such as UTF-8 and UTF-16; other times it identifies a

different character set such as ISO-8859-1 or US-ASCII, which for XML's purposes serves mainly as an encoding of a subset of the full Unicode character set.

The default encoding is UTF-8 if no encoding declaration or other metadata is present. UTF-16 can also be used if the document begins with a byte order mark. However, even in cases where the document is written in the UTF-8 or UTF-16 encodings, an `encoding` declaration helps people reading the document recognize the encoding, so it's useful to specify it explicitly.

Try to stick to well-known standard character sets and encodings such as ISO-8859-1, UTF-8, and UTF-16 if possible. You should always use the standard names for these character sets. Table 1–1 lists the names defined by the XML 1.0 specification. All parsers that support these character sets should recognize these names. For character encodings not defined in XML 1.0, choose a name registered with the IANA. You can find a complete list at http://www.iana.org/assignments/character-sets/. However, you should avoid nonstandard names. In particular, watch out for Java names like 8859_1 and UTF16. Relatively few parsers not written in Java recognize these, and even some Java parsers don't recognize them by default. However, all parsers including those written in Java should recognize the IANA standard equivalents such as ISO-8859-1 and UTF-16.

For similar reasons, avoid declaring and using vendor-dependent character sets such as Cp1252 (U.S. Windows) or MacRoman. These are not as interoperable as the standard character sets across the heterogeneous set of platforms that XML supports.

The `standalone` Declaration

The `standalone` attribute has the value yes or no. If no `standalone` declaration is present, then no is the default.

A yes value means that no declarations in the external DTD subset affect the content of the document in any way. Specifically, the following four conditions apply.

1. No default attribute values are specified for elements.
2. No entity references used in the instance document are defined.
3. No attribute values need to be normalized.
4. No elements contain ignorable white space.

Table 1–1 | Character Set Names Defined in XML

Name	Set
UTF-8	Variable width, byte order independent Unicode
UTF-16	Two-byte Unicode with surrogate pairs
ISO-10646-UCS-2	Two-byte Unicode without surrogate pairs; plane 0 only
ISO-10646-UCS-4	Four-byte Unicode
ISO-8859-1	Latin-1; mostly compatible with the standard U.S. Windows character set
ISO-8859-2	Latin-2
ISO-8859-3	Latin-3
ISO-8859-4	Latin-4
ISO-8859-5	ASCII and Cyrillic
ISO-8859-6	ASCII and Arabic
ISO-8859-7	ASCII and Greek
ISO-8859-8	ASCII and Hebrew
ISO-8859-9	Latin-5
ISO-8859-10	Latin-6
ISO-8859-11	ASCII plus Thai
ISO-8859-13	Latin-7
ISO-8859-14	Latin-8
ISO-8859-15	Latin-9, Latin-0
ISO-8859-16	Latin-10
ISO-2022-JP	A combination of ISO 646 (a slight variant of ASCII) and JIS X0208 that uses escape sequences to switch between the two character sets
Shift_JIS	A combination of JIS X0201:1997 and JIS X0208:1997 that uses escape sequences to switch between the two character sets
EUC-JP	A combination of four code sets (ASCII, JIS X0208-1990, half width Katakana, and JIS X0212-1990) that uses escape sequences to switch between the character sets

If these conditions hold, the parser may choose not to read the external DTD subset, which can save a significant amount of time when the DTD is at a remote and slow web site.

A nonvalidating parser will not actually check that these conditions hold. For example, it will not report an error if an element does not have an attribute for which a default value is provided in the external DTD subset. Obviously the parser can't find mistakes that are apparent only when it reads the external DTD subset if it doesn't read the external DTD subset.

A validating parser is supposed to report a validity error if `standalone` has the value yes and any of these four conditions are not true.

It is always acceptable to set `standalone` to no, even if the document could technically stand alone. If you don't want to be bothered figuring out whether all of the above four conditions apply, just set `standalone="no"` (or leave it unspecified because the default is no). This is always correct.

The `standalone` declaration only applies to content read from the external DTD subset. It has nothing to do with other means of merging in content from remote documents such as schemas, XIncludes, XLinks, application-specific markup like the `img` element in XHTML, or anything else. It is strictly about the DTD.

Whatever values you pick for the `version`, `encoding`, and `standalone` attributes, and whether you include `encoding` and `standalone` attributes at all, you should provide an XML declaration. It only takes a few bytes and makes it much easier for both people and parsers to process your document.

Item 2 Mark Up with ASCII if Possible

Despite the rapid growth of Unicode in the last few years, the sad fact is that many text editors and other tools are still tied to platform- and nationality-dependent character sets such as Windows-1252, Mac-Roman, and SJIS. The only characters all these sets have in common are the 128 ASCII letters, digits, punctuation marks, and control characters. These characters are the only ones that can be reliably displayed and edited across the wide range of computers and software in use today. Thus, if it's not too big a problem, try to limit your markup to the ASCII character set. If you're writing in English, this is normally not a problem.

On the other hand, this principle is not written in stone, especially if you're not working in English. If you're writing a simple vocabulary for a local French bank without any international ambitions, you will probably want to include all the accents commonly used in French for words like *relevé* (statement) and *numéro* (number). For instance, a bank statement might look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Relevé xmlns="http://namespaces.petitebanque.com/">
```

```
  <Banque>PetiteBanque</Banque>
  <Compte>
    <Numéro>00003145298</Numéro>
    <Type>épargne</Type>
    <Propriétaire>Jean Deaux</Propriétaire>
  </Compte>
  <Date>2003-30-02</Date>
  <SoldeDOuverture>5266.34</SoldeDOuverture>
  <Transaction type="dépôt">
    <Date>2003-02-07</Date>
    <Somme>300.00</Somme>
  </Transaction>
  <Transaction type="transfert">
    <Compte>
      <Numéro>0000271828</Numéro>
      <Type>courant</Type>
      <Propriétaire>Jean Deaux</Propriétaire>
    </Compte>
    <Date>2003-02-07</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="dépôt">
    <Date>2003-02-15</Date>
    <Somme>512.32</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-15</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-25</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <SoldeDeFermeture>5478.64</SoldeDeFermeture>
</Relevé>
```

However, this code is likely to cause trouble if the document ever crosses national or linguistic boundaries. For instance, if programmers at the bank's Athens branch open the same document in a text editor, they're likely to see something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Relevι xmlns="http://namespaces.petitebanque.com/">
  <Banque>PetiteBanque</Banque>
  <Compte>
    <Numιro>00003145298</Numιro>
    <Type>ιpargne</Type>
    <Propriιtaire>Jean Deaux</Propriιtaire>
  </Compte>
  <Date>2003-30-02</Date>
  <SoldeDOuverture>5266.34</SoldeDOuverture>
  <Transaction type="dιptt">
    <Date>2003-02-07</Date>
    <Somme>300.00</Somme>
  </Transaction>
  <Transaction type="transfert">
    <Compte>
      <Numιro>0000271828</Numιro>
      <Type>courant</Type>
      <Propriιtaire>Jean Deaux</Propriιtaire>
    </Compte>
    <Date>2003-02-07</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="dιptt">
    <Date>2003-02-15</Date>
    <Somme>512.32</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-15</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-25</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <SoldeDeFermeture>5478.64</SoldeDeFermeture>
</Relevι>
```

The e's with accents acute have morphed into iotas, and the o's with carets have turned into lowercase taus.

Indeed, even crossing platform boundaries within the same country may cause problems. Were the same document opened on a Mac, the developers would likely see something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<RelevÈ xmlns="http://namespaces.petitebanque.com/">
  <Banque>PetiteBanque</Banque>
  <Compte>
    <NumÈro>00003145298</NumÈro>
    <Type>Èpargne</Type>
    <PropriÈtaire>Jean Deaux</PropriÈtaire>
  </Compte>
  <Date>2003-30-02</Date>
  <SoldeDOuverture>5266.34</SoldeDOuverture>
  <Transaction type="dÈpÙt">
    <Date>2003-02-07</Date>
    <Somme>300.00</Somme>
  </Transaction>
  <Transaction type="transfert">
    <Compte>
      <NumÈro>0000271828</NumÈro>
      <Type>courant</Type>
      <PropriÈtaire>Jean Deaux</PropriÈtaire>
    </Compte>
    <Date>2003-02-07</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="dÈpÙt">
    <Date>2003-02-15</Date>
    <Somme>512.32</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-15</Date>
    <Somme>200.00</Somme>
  </Transaction>
  <Transaction type="retrait">
    <Date>2003-02-25</Date>
    <Somme>200.00</Somme>
  </Transaction>
```

```
  <SoldeDeFermeture>5478.64</SoldeDeFermeture>
</RelevÈ>
```

In this case, the lowercase e's with accents acute have changed to uppercase E's with accents grave, and the accented o's have also changed. This isn't quite as bad, but it's more than enough to confuse most software applications and not a few people.

The encoding declaration fixes these issues for XML-aware tools such as parsers and XML editors. However, it doesn't help non-XML-aware systems like plain text editors and regular expressions. (See Item 29.) There's a lot of flaky code out there in the world in less than perfect systems.

Using Unicode instead of ISO-8859-1 for the character set goes a long way toward fixing this particular problem. (See Item 38.) However, this opens up several lesser but still significant problems.

1. Many text editors can't handle Unicode. Even editors that can often have trouble recognizing Unicode documents.
2. The only glyphs (graphical representations of characters from a particular font) you can rely on being available across a variety of systems cover the ASCII range. Even a system that can process all Unicode characters may not be able to display them.
3. Keyboards don't have the right keys for more than a few languages. The basic ASCII characters are the only characters likely to be available anywhere in the world. Indeed, even a few ASCII characters are problematic. I once purchased a French keyboard in Montreal that did not have a single quote key.
4. The string facilities of many languages and operating systems implicitly assume single-byte characters. This includes the char data type in C. Java is slightly better but still can't handle all Unicode characters.

None of these problems are insurmountable. Programmers' editors that properly handle Unicode are available for almost all systems of interest. You can purchase or download fonts that cover most Unicode blocks, though you may have to mix and match several fonts to get full coverage. Input methods, multi-key combinations, and graphical keymaps allow authors to type accented characters and ideographic characters even on U.S. keyboards. It is possible to write Unicode-savvy Java, C, and Perl code provided you have a solid understanding of Unicode and know exactly

where those languages' usual string and character types are inadequate. Just be aware that if you do use non-ASCII characters for your markup, these issues will arise.

One final caveat: I am primarily concerned with *markup* here, that is, element and attribute names. I am not talking about element content and attribute values. To the extent that such content is written in a natural human language, that content really needs to be written in that language with all its native characters intact. For instance, if a customer's name is Thérèse Barrière, it should be written as Thérèse Barrière, not Therese Barriere. (In some jurisdictions there are even laws requiring this.) Non-ASCII content raises many of the same issues as non-ASCII markup. However, the need for non-ASCII characters is greater here, and the problems aren't quite as debilitating.

While the situation is improving slowly, for the time being, documents will be more easily processed in an international, heterogeneous environment if they contain only ASCII characters. ASCII is a lowest common denominator and a very imperfect one at that. However, it is the lowest common denominator. In the spirit of being liberal in what you accept but conservative in what you generate, you should use ASCII when possible. If the text you're marking up is written in any language other than English, you'll almost certainly have to use other character sets. Just don't choose to do so gratuitously. For example, don't pick ISO-8859-1 (Latin-1) just so you can tag a curriculum vitae with `<resumé>` instead of `<resume>`.

Item 3 Stay with XML 1.0

Everything you need to know about XML 1.1 can be summed up in two rules.

1. Don't use it.
2. (For experts only) If you speak Mongolian, Yi, Cambodian, Amharic, Dhivehi, Burmese, or a very few other languages and you want to write your markup (not your text but your markup) in these languages, you can set the `version` attribute of the XML declaration to 1.1. Otherwise, refer to rule 1.

XML 1.1 does several things, one of them marginally useful to a few developers, the rest actively harmful.

- It expands the set of characters allowed as name characters.
- The C0 control characters (except for NUL) such as form feed, vertical tab, BEL, and DC1 through DC4 are now allowed in XML text provided they are escaped as character references.
- The C1 control characters (except for NEL) must now be escaped as character references.
- NEL can be used in XML documents but is resolved to a line feed on parsing.
- Parsers may (but do not have to) tell client applications that Unicode data was not normalized.
- Namespace prefixes can be undeclared.

Let's look at these changes in more detail.

New Characters in XML Names

XML 1.1 expands the set of characters allowed in XML names (that is, element names, attribute names, entity names, ID-type attribute values, and so forth) to allow characters that were not defined in Unicode 2.0, the version that was extant when XML 1.0 was first defined. Unicode 2.0 is fully adequate to cover the needs of markup in English, French, German, Russian, Chinese, Japanese, Spanish, Danish, Dutch, Arabic, Turkish, Hebrew, Farsi, Thai, Hindi, and most other languages you're likely to be familiar with as well as several thousand you aren't. However, Unicode 2.0 did miss a few important living languages including Mongolian, Yi, Cambodian, Amharic, Dhivehi, and Burmese, so if you want to write your markup in these languages, XML 1.1 is worthwhile.

However, note that this is relevant only if we're talking about *markup,* particularly element and attribute names. It is not necessary to use XML 1.1 to write XML data, particularly element content and attribute values, in these languages. For example, here's the beginning of an Amharic translation of the Book of Matthew written in XML 1.0.

```
<?xml version="1.0" encoding="UTF-8">
<book>
<title>የማቴዎስ ወንጌል</title>
<chapter number="፩">
<title>የኢየሱስ የትው ልድ ሐረግ</title>
<verse number="፩">
 የዳዊት ልጅ፥ የአከርሃም ልጅ የሁ ነው የኢየሱስ ክርስቶስ የትው ልድ ሐረግ የሚከተለው ነው፤
```

```
</verse>
<verse number="፪">
 ኣከርሃም ይስሐቅን ወለደ፤
 ይስሐቅ ያዕቆብን ወለደ፤
 ያዕቆብ ይሁዳንና ወንድሞቹን ወለደ፤
</verse>
</chapter>
</book>
```

Here the element and attribute names are in English although the content and attribute values are in Amharic. On the other hand, if we were to write the element and attribute names in Amharic, we would need to use XML 1.1.

```
<?xml version="1.1" encoding="UTF-8">
```

```
<መጽሐፉ>
<ኣርእስት>የማቴዎስ ወንጌል</ኣርእስት>
<ምዕራፉ ቁጥር="፩">
<ኣርእስት>የኢየሱስ የትው ልድ ሐረግ</ኣርእስት>
<ቤት ቁጥር="፩">
የዳዊት ልጅ፥ የኣከርሃም ልጅ የሁ ነው የኢየሱስ ከርስቶስ የትው ልድ ሐረግ የሚከተለው ነው፤
</ቤት>
<ቤት ቁጥር="፪">
 ኣከርሃም ይስሐቅን ወለደ፤
 ይስሐቅ ያዕቆብን ወለደ፤
 ያዕቆብ ይሁዳንና ወንድሞቹን ወለደ፤
</ቤት>
</ምዕራፉ>
</መጽሐፉ>
```

This is plausible. A native Amharic speaker might well want to write markup like this. However, the loosening of XML's name character rules have effects far beyond the few extra languages they're intended to enable. Whereas XML 1.0 is conservative (everything not permitted is forbidden),

XML 1.1 is liberal (everything not forbidden is permitted). XML 1.0 lists the characters you can use in names. XML 1.1 lists the characters you can't use in names. Characters XML 1.1 allows in names include:

- Symbols like the copyright sign (©)
- Mathematical operators such as ±
- Superscript 7 (^{7})
- The musical symbol for a six-string fretboard
- The zero-width space
- Private-use characters
- Several hundred thousand characters that aren't even defined in Unicode and probably never will be

XML 1.1's lax name character rules have the potential to make documents much more opaque and obfuscated.

C0 Control Characters

The first 32 Unicode characters with code points from 0 to 31 are known as the C0 controls. They were originally defined in ASCII to control teletypes and other monospace dumb terminals. Aside from the tab, carriage return, and line feed they have no obvious meaning in text. Since XML is text, it does not include binary characters such as NULL (#x00), BEL (#x07), DC1 (#x11) through DC4 (#x14), and so forth. These noncharacters are historical relics. XML 1.0 does not allow them. This is a good thing. Although dumb terminals and binary-hostile gateways are far less common today than they were twenty years ago, they are still used, and passing these characters through equipment that expects to see plain text can have nasty consequences, including disabling the screen. (One common problem that still occurs is accidentally paging a binary file on a console. This is generally quite ugly and often disables the console.)

A few of these characters occasionally do appear in non-XML text data. For example, the form feed (#x0C) is sometimes used to indicate a page break. Thus moving data from a non-XML system such as a BLOB or CLOB field in a database into an XML document can unexpectedly cause malformedness errors. Text may need to be cleaned before it can be added to an XML document. However, the far more common problem is that a document's encoding is misidentified, for example, defaulted as UTF-8 when it's really UTF-16 or ISO-8859-1. In this case, the parser will notice unexpected nulls and throw a well-formedness error.

XML 1.1 fortunately still does not allow raw binary data in an XML document. However, it does allow you to use character references to escape the C0 controls such as form feed and BEL. The parser will resolve them into the actual characters before reporting the data to the client application. You simply can't include them directly. For example, the following document uses form feeds to separate pages.

```
<?xml version="1.1">
<book>
  <title>Nursery Rhymes</title>
  <rhyme>
    <verse>Mary, Mary quite contrary</verse>
    <verse>How does your garden grow?</verse>
  </rhyme>
  &#x0C;
  <rhyme>
    <verse>Little Miss Muffet sat on a tuffet</verse>
    <verse>Eating her curds and whey</verse>
  </rhyme>
  &#x0C;
  <rhyme>
    <verse>Old King Cole was a merry old soul</verse>
    <verse>And a merry old soul was he</verse>
  </rhyme>
</book>
```

However, this style of page break died out with the line printer. Modern systems use stylesheets or explicit markup to indicate page boundaries. For example, you might place each separate page inside a `page` element or add a `pagebreak` element where you wanted the break to occur, as shown below.

```
<?xml version="1.1">
<book>
  <title>Nursery Rhymes</title>
  <rhyme>
    <verse>Mary, Mary quite contrary</verse>
    <verse>How does your garden grow?</verse>
  </rhyme>
  <pagebreak/>
  <rhyme>
```

```
    <verse>Little Miss Muffet sat on a tuffet</verse>
    <verse>Eating her curds and whey</verse>
  </rhyme>
  <pagebreak/>
  <rhyme>
    <verse>Old King Cole was a merry old soul</verse>
    <verse>And a merry old soul was he</verse>
  </rhyme>
</book>
```

Better yet, you might not change the markup at all, just write a stylesheet that assigns each rhyme to a separate page. Any of these options would be superior to using form feeds. Most uses of the other C0 controls are equally obsolete.

There is one exception. You still cannot embed a null in an XML document, not even with a character reference. Allowing this would have caused massive problems for C, C++, and other languages that use null-terminated strings. The null is still forbidden, even with character escaping, which means it's still not possible to directly embed binary data in XML. You have to encode it using Base64 or some similar format first. (See Item 19.)

C1 Control Characters

There is a less common block of C1 control characters between 128 (#x80) and 159 (#x9F). These include start of string, end of string, cancel character, privacy message, and a few other equally obscure characters. For the most part these are even less useful and less appropriate for XML documents than the C0 control characters. However, they were allowed in XML 1.0 mostly by mistake. XML 1.1 rectifies this error (with one notable exception, which I'll address shortly) by requiring that these control characters be escaped with character references as well. For example, you can no longer include a "break permitted here" character in element content or attribute values. You have to write it as `&#x82;` instead.

This actually does have one salutary effect. There are a lot of documents in the world that are labeled as ISO-8859-1 but actually use the nonstandard Microsoft Cp1252 character set instead. Cp1252 does not include the C1 controls. Instead it uses this space for extra graphic characters such as €, Œ, and ™. This causes significant interoperability problems when

moving documents between Windows and non-Windows systems, and these problems are not always easy to detect.

By making escaping of the C1 controls mandatory, such mislabeled documents will now be obvious to parsers. Any document that contains an unescaped C1 character labeled as ISO-8859-1 is malformed. Documents that correctly identify themselves as Cp1252 are still allowed.

The downside to this improvement is that there is now a class of XML documents that is well-formed XML 1.0 but not well-formed XML 1.1. XML 1.1 is not a superset of XML 1.0. It is neither forward nor backward compatible.

NEL Used as a Line Break

The fourth change XML 1.1 makes is of no use to anyone and should never have been adopted. XML 1.1 allows the Unicode next line character (#x85, NEL) to be used anywhere a carriage return, line feed, or carriage return–line feed pair is used in XML 1.0 documents. Note that a NEL doesn't mean anything different than a carriage return or line feed. It's just one more way of adding extra white space. However, it is incompatible not only with the installed base of XML software but also with all the various text editors on UNIX, Windows, Mac, OS/2, and almost every other non-IBM platform on Earth. For instance, you can't open an XML 1.1 document that uses NELs in emacs, vi, BBEdit, UltraEdit, jEdit, or most other text editors and expect it to put the line breaks in the right places. Figure 3–1 shows what happens when you load a NEL-delimited file into emacs. Most other editors have equal or bigger problems, especially on large documents.

If so many people and platforms have such problems with NEL, why has it been added to XML 1.1? The problem is that there's a certain huge monopolist of a computer company that doesn't want to use the same standard everyone else in the industry uses. And—surprise, surprise—its name isn't Microsoft. No, this time the villain is IBM. Certain IBM mainframe software, particularly console-based text editors like XEdit and OS/390 C compilers, do not use the same two line-ending characters (carriage return and line feed) that everybody else on the planet has been using for at least the last twenty years. Instead those text editors use character #x85, NEL.

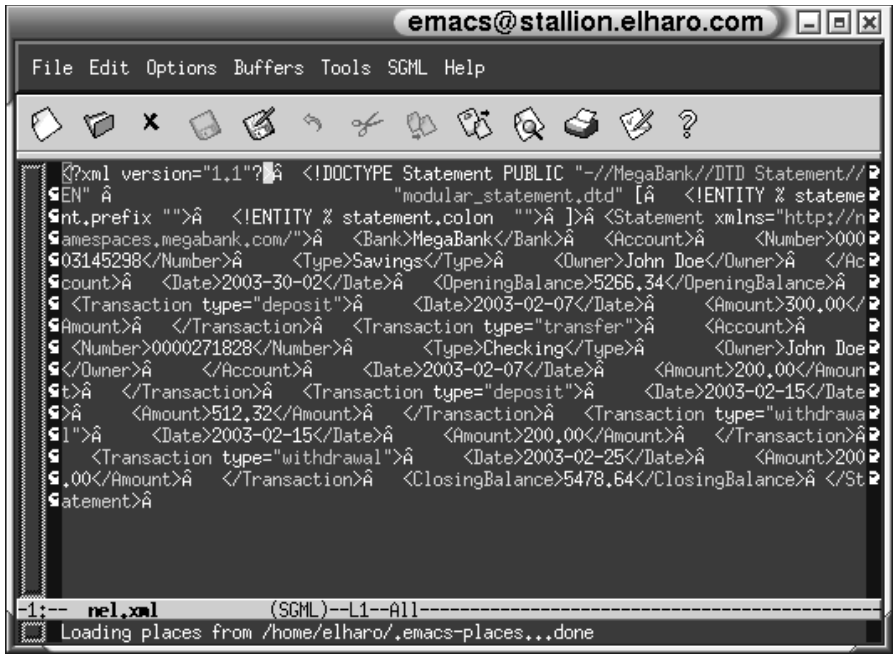

Figure 3–1 | Loading a NEL-Delimited File into a Non-IBM Text Editor

If you're one of those few developers writing XML by hand with a plain console editor on an IBM mainframe, you should upgrade your editor to support the line-ending conventions the rest of the world has standardized on. If you're writing C code to generate XML documents on a mainframe, you just need to use \x0A instead of \n to represent the line end. (Java does not have this problem.) If you're reading XML documents, the parser should convert the line endings for you. There's no need to use XML 1.1.

Unicode Normalization

For reasons of compatibility with legacy character sets such as ISO-8859-1 (as well as occasional mistakes) Unicode sometimes provides multiple representations of the same character. For example, the e with accent acute (é) can be represented as either the single character #xE9 or with the two characters #x65 (e) followed by #x301 (combining accent acute). XML 1.1 suggests that all generators of XML text should normalize such

alternatives into a canonical form. In this case, you should use the single character rather than the double character.

However, both forms are still accepted. Neither is malformed. Furthermore, parsers are explicitly prohibited from doing the normalization for the client program. They may merely report a nonfatal error if the XML is found to be unnormalized. In fact, this is nothing that parsers couldn't have done with XML 1.0, except that it didn't occur to anyone to do it. Normalization is more of a strongly recommended best practice than an actual change in the language.

Undeclaring Namespace Prefixes

There's one other new feature that's effectively part of XML 1.1: namespaces 1.1, which adds the ability to undeclare namespace prefix mappings. For example, consider the following API element.

```
<?xml version="1.0" encoding="UTF-8">
<API xmlns:public="http://www.example.com"
      xmlns:private="http://www.example.org" >
  <title>Geometry</title>
  <cpp xmlns:public="" xmlns:private="">
    class CRectangle {
      int x, y;
      public:void set_values (int,int);
      private:int area (void); }
  </cpp>
</API>
```

A system that was looking for qualified names in element content might accidentally confuse the `public:void` and `private:int` in the `cpp` element with qualified names instead of just part of C++ syntax (albeit ugly C++ syntax that no good programmer would write). Undeclaring the public and private prefixes allows them to stand out for what they actually are, just plain unadorned text.

In practice, however, very little code looks for qualified names in element content. Some code does look for these things in attribute values, but in those cases it's normally clear whether or not a given attribute can contain qualified names. Indeed this example is so forced precisely because prefix undeclaration is very rarely needed in practice and never needed if you're only using prefixes on element and attribute names.

That's it. There is nothing else new in XML 1.1. It doesn't move namespaces or schemas into the core. It doesn't correct admitted mistakes in the design of XML such as attribute value normalization. It doesn't simplify XML by removing rarely used features like unparsed entities and notations. It doesn't even clear up the confusion about what parsers should and should not report. All it does is change the list of name and white space characters. This very limited benefit comes at an extremely high cost. There is a huge installed base of XML 1.0–aware parsers, browsers, databases, viewers, editors, and other tools that don't work with XML 1.1. They will report well-formedness errors when presented with an XML 1.1 document.

The disadvantages of XML 1.1 (including the cost in both time and money of upgrading all your software to support it) are just too great for the extremely limited benefits it provides most developers. If you're more comfortable working in Mongolian, Yi, Cambodian, Amharic, Dhivehi, or Burmese and you only need to exchange data with other speakers of one of these languages (for instance, you're developing a system exclusively for a local Amharic-language newspaper in Addis Ababa where everybody speaks Amharic), you can set the `version` attribute of the XML declaration to 1.1. Everyone else should stick to XML 1.0.

Item 4 Use Standard Entity References

No reasonably sized keyboard could possibly include all the characters in Unicode. U.S. keyboards are especially weak when it comes to typing in foreign languages with unusual accents and non-Latin scripts. XML allows you to use either character references or entity references to address this problem. In general, named entity references like `&Ecaron;` should be preferred to character references like `&#x011B;` because they're easier on the human beings who have to read the source code.

While there are no standards for how to name these entity references, there are some useful entity sets bundled with XHTML, DocBook, and MathML. Since these are all modular specifications, you can even use the DTDs that define their entity sets without pulling in the rest of the application. For example, if you just want to use the standard HTML entity references many designers have already memorized like `&copy;` and ` `, you could add the following lines to your DTD.

```
<!ENTITY nbsp   " ">
<!ENTITY iexcl  "&#161;">
<!ENTITY cent   "&#162;">
<!ENTITY pound  "&#163;">
<!ENTITY curren "&#164;">
<!ENTITY yen    "&#165;">
<!ENTITY brvbar "&#166;">
<!ENTITY sect   "&#167;">
<!ENTITY uml    "&#168;">
<!ENTITY copy   "&#169;">
...
```

Better yet, you could store local copies of the relevant DTDs in the same directory as your own DTD and just point to them.

```
<!ENTITY % HTMLlat1 PUBLIC
   "-//W3C//ENTITIES Latin 1 for XHTML//EN"
   "xhtml-lat1.ent">
%HTMLlat1;

<!ENTITY % HTMLsymbol PUBLIC
   "-//W3C//ENTITIES Symbols for XHTML//EN"
   "xhtml-symbol.ent">
%HTMLsymbol;

<!ENTITY % HTMLspecial PUBLIC
   "-//W3C//ENTITIES Special for XHTML//EN"
   "xhtml-special.ent">
%HTMLspecial;
```

If you're using catalogs (Item 47), you can use the public IDs to locate the local caches of these entity sets.

Even if you're defining your own entity references for a particular subset of Unicode, I still suggest using the standard names. The HTML names are far and away the most popular, so if there's an HTML name for a certain character, by all means use it. For example, I would never call Unicode character 0xA0, the nonbreaking space, anything other than ` `.

If HTML does not have a standard name for a character, I normally turn to DocBook next. Its entity names are based on the standard SGML entity names. (These are among the SGML features that got dropped out of

XML in the process of making it simple enough for mere mortals to use.) They aren't as well known as the HTML entity names, but they are much more comprehensive and are an international standard. The SGML entity names include those listed below.

ISO 8879:1986//ENTITIES Added Math Symbols: Arrow Relations//EN//XML
ISO 8879:1986//ENTITIES Added Math Symbols: Binary Operators//EN//XML
ISO 8879:1986//ENTITIES Added Math Symbols: Delimiters//EN//XML
ISO 8879:1986//ENTITIES Added Math Symbols: Negated Relations//EN//XML
ISO 8879:1986//ENTITIES Added Math Symbols: Ordinary//EN//XML
ISO 8879:1986//ENTITIES Added Math Symbols: Relations//EN//XML
ISO 8879:1986//ENTITIES Box and Line Drawing//EN//XML
ISO 8879:1986//ENTITIES Russian Cyrillic//EN//XML
ISO 8879:1986//ENTITIES Non-Russian Cyrillic//EN//XML
ISO 8879:1986//ENTITIES Diacritical Marks//EN//XML
ISO 8879:1986//ENTITIES Greek Letters//EN//XML
ISO 8879:1986//ENTITIES Monotoniko Greek//EN//XML
ISO 8879:1986//ENTITIES Greek Symbols//EN//XML
ISO 8879:1986//ENTITIES Alternative Greek Symbols//EN//XML
ISO 8879:1986//ENTITIES Added Latin 1//EN//XML
ISO 8879:1986//ENTITIES Added Latin 2//EN//XML
ISO 8879:1986//ENTITIES Numeric and Special Graphic//EN//XML
ISO 8879:1986//ENTITIES Publishing//EN//XML
ISO 8879:1986//ENTITIES General Technical//EN//XML

Finally, for mathematically oriented characters like ∮ and ⊕, I turn to MathML 2.0. Its entity sets, found at http://www.w3.org/TR/MathML2/chapter6.html, cover more of the special characters that Unicode 3.0 and later define for mathematics than the pure SGML mathematical entity sets.

Item 5 Comment DTDs Liberally

DTDs can be as obfuscated as C++ code written by a first-year undergrad. You should use lots of comments to explain exactly what's going on. Well-written DTDs such as the modular XHTML DTD often contain more than twice as many lines of comments as actual code. The list that follows shows some of the information you should include in a DTD.

- Who wrote it and how to contact them
- The copyright and use conditions that apply to the DTD
- The version of the DTD
- The namespace URI
- The PUBLIC and SYSTEM identifiers for the DTD
- The root elements
- A brief description of the application the DTD models
- The purpose and content of each parameter entity defined
- The purpose and content of each general entity defined
- The meaning of each element declared by an ELEMENT declaration
- The meaning of each attribute declared in an ATTLIST declaration
- The meaning of each notation declared in a NOTATION declaration
- Additional constraints on elements and attributes that cannot be specified in a DTD, for example, that a length attribute must contain a positive integer

For example, let's consider a DTD designed for bank statements like the ones your bank sends you at the end of every month. If the account isn't very active a typical document might look like this:

```
<?xml version="1.0"?>
<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "statement.dtd">
<Statement xmlns="http://namespaces.megabank.com/">
  <Bank>MegaBank</Bank>
  <Account>
    <Number>00003145298</Number>
    <Type>Savings</Type>
    <Owner>John Doe</Owner>
  </Account>
  <Date>2003-30-02</Date>
  <OpeningBalance>5266.34</OpeningBalance>
  <Transaction type="deposit">
    <Date>2003-02-07</Date>
    <Amount>300.00</Amount>
  </Transaction>
  <Transaction type="transfer">
    <Account>
      <Number>0000271828</Number>
      <Type>Checking</Type>
      <Owner>John Doe</Owner>
```

```
    </Account>
    <Date>2003-02-07</Date>
    <Amount>200.00</Amount>
  </Transaction>
  <Transaction type="deposit">
    <Date>2003-02-15</Date>
    <Amount>512.32</Amount>
  </Transaction>
  <Transaction type="withdrawal">
    <Date>2003-02-15</Date>
    <Amount>200.00</Amount>
  </Transaction>
  <Transaction type="withdrawal">
    <Date>2003-02-25</Date>
    <Amount>200.00</Amount>
  </Transaction>
  <ClosingBalance>5478.64</ClosingBalance>
</Statement>
```

You already saw this example in French in Item 2. We'll be looking at different aspects of it in several later items too, but for now let's think about what's appropriate for the DTD.

The Header Comment

The DTD should start with one long comment that lists lots of metadata about the DTD. Generally, this would start with a title specifying the XML application the DTD describes. For example:

```
MegaBank Account Statement DTD, Version 1.1.2
```

This would normally be followed with some copyright notice. For example:

```
Copyright 2003 MegaBank
```

Alternately, you could use the copyright symbol:

```
© 2003 MegaBank
```

Do **not** use a *c* inside parentheses:

```
(c) 2003 MegaBank
```

This is not recognized by the international treaty that establishes copyright law in most countries. For similar reasons do not use both the word *Copyright* and the symbol like this:

```
Copyright © 2003 MegaBank
```

Neither of these forms is legally binding. I wouldn't want to rely on the difference between © and (c) in a defense against a claim of copyright infringement, but as a copyright owner I wouldn't want to count on them being considered the same either. If the genuine symbol is too hard to type because you're restricted to ASCII, just use the word *Copyright* with a capital *C* instead.

Now that the DTD has been copyrighted, the next question is, who do you want to allow to use the DTD and under what conditions? If the DTD is purely for internal use and you don't want to allow anyone else to use it, a simple `© 2003 MegaBank` statement may be all you need. However, by default such a statement makes the DTD at least technically illegal for anyone else to use, copy, or distribute without explicit permission from the copyright owner. If it is your intention that this DTD be used by multiple parties, you should include language explicitly stating that. For example, this statement would allow third parties to reuse the DTD but not to modify it:

```
This DTD may be freely copied and redistributed.
```

If you want to go a little further, you can allow other parties to modify the DTD. For example, one of the most liberal licenses common among DTDs is the W3C's, which states:

```
Permission to use, copy, modify and distribute the XHTML
Basic DTD and its accompanying documentation for any purpose
and without fee is hereby granted in perpetuity, provided
that the above copyright notice and this paragraph appear in
all copies. The copyright holders make no representation
about the suitability of the DTD for any purpose.
```

Often some authorship information is also included. As well as giving credit where credit is due (or assigning blame), this is important so that users know who they can ask questions of or report bugs to. Depending on circumstances the contact information may take different forms. For instance, a private DTD inside a small company might use a personal name and a phone extension while a large public DTD might provide the

name and URL of a committee. Whichever form it takes, there should be some means of contacting a party responsible for the DTD. For example:

```
Prepared by the MegaBank Interdepartment XML Committee
Joseph Quackenbush, editor <jquackenbush@megabank.com>
http://xml.megabank.com/
```

If you modify the DTD, you should add your name and contact information and indicate how the DTD has been modified. However, you should also retain the name of the original author. For example:

```
International Statement DTD prepared for MegaBank France to
satisfy EEC banking regulations by Stefan Hilly
<shilly@megabank.fr>

Original prepared by the MegaBank Interdepartment XML Committee
Joseph Quackenbush, editor <jquackenbush@megabank.com>
http://xml.megabank.com/
```

Following the copyright and authorship information, the next thing is normally a brief description of the XML application the DTD describes. For example, the bank statement application might include something like this:

```
This is the DTD for MBSML, the MegaBank Statement Markup
Language. It is used for account statements sent to both
business and consumer customers at the end of each month.
Each document represents a complete statement for a single
account. It handles savings, checking, CD, and money market
accounts. However, it is not used for credit cards or loans.
```

This is often followed by useful information about the DTD that is not part of the DTD grammar itself. For example, the following comment describes the namespace URI, the root element, the public ID, and the customary system ID for this DTD.

```
All elements declared by this DTD are in the
http://namespaces.megabank.com/statement namespace.

Documents adhering to this DTD should have the root element
Statement.

This DTD is identified by these PUBLIC and SYSTEM identifiers:

   PUBLIC "-//MegaBank//DTD Statement//EN"
   SYSTEM "http://dtds.megabank.com/statement.dtd"
```

```
The system ID may be pointed at a local copy of the DTD
instead. For example,

<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "statement.dtd">

The internal DTD subset should *not* be used to customize
the DTD.
```

Some DTDs also include usage instructions or detailed lists of changes in the current version. There's certainly nothing wrong with this, but I normally prefer to point to some canonical source of documentation for the application. For example:

```
For more information see http://xml.megabank.com/statement/
```

This pretty much exhausts the information that's customarily stored in the header.

Declarations

Next come the individual declarations, and in well-commented DTDs each such declaration should also have a comment expanding on it in more detail. Let's begin with a simple element declaration, such as this one for an amount:

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

All the declaration tells us is that the amount contains text and no child elements. It does not say whether this is an amount of money, an amount of time, an amount of oranges, or anything else. According to this declaration all of the following are valid:

```
<Amount>$3.45</Amount>
<Amount>3.45</Amount>
<Amount>-3.45</Amount>
<Amount>(3.45)</Amount>
<Amount>3.45 EUR</Amount>
<Amount>+3.45 EUR</Amount>
<Amount>23</Amount>
<Amount />
<Amount>Oh Susanna, won't you sing a song for me?</Amount>
```

Naturally, the application likely has somewhat more rigid requirements than this. Although you can't enforce these requirements with a DTD, it is a very good idea to list them in a comment like this:

```
<!-- An amount element contains an amount of money, using two
     decimal places of precision for decimalized currencies;
     e.g.,

       <Amount>3.45</Amount>

     Nondecimalized currencies such as yen are given as
     an integer; e.g.

       <Amount>3450</Amount>

     A leading minus sign is used to indicate a negative
     number.
     White space is not allowed. -->
<!ELEMENT Amount (#PCDATA)>
```

The same is true for attribute values. For example, consider this `ATTLIST` declaration that declares a `currency` attribute. The type is `NMTOKEN`, but what else can be said about it?

```
<!ATTLIST Amount currency NMTOKEN "USD">
```

The following comment makes it clear that the value of the `currency` attribute is a three-letter ASCII currency code as defined by ISO 4217:2001 *Codes for the Representation of Currencies and Funds*. The default value can be read from the `ATTLIST` declaration, but it doesn't hurt to be more explicit.

```
<!-- Currencies are given as one of the three-letter codes
     defined by ISO 4217. Codes for countries where we have
     branches are:

     AUD  Australian Dollar
     BRL  Brazilian Real
     BSD  Bahaman Dollar
     CAD  Canadian Dollar
     CHF  Swiss Franc
     DKK  Danish Krone
     EUR  European Currency Unit (Euro, formerly known as ECU)
     GBP  British Pound
     JPY  Japanese Yen
```

```
     KYD  Cayman Dollar
     MXN  Mexican Peso (new)
     MXP  Mexican Peso (old)
     USD  American Dollar

      The complete list can be found at
      http://www.gnucash.org/docs/C/xacc-isocurr.html.

      The default currency is U.S. dollars.
 -->
```

Sometimes when one type of value, such as money or currency, is going to be used for multiple elements in the DTD, it's useful to define it as a named parameter entity that can be referenced from different locations in the DTD. In this case, the comment moves on to discuss the parameter entity.

```
<!-- A money element uses two decimal places of precision for
     decimalized currencies and an integer for nondecimalized
     currencies. Noninteger, nondecimal currencies are not
     supported. A leading minus sign is used to indicate a
     negative number. White space is not allowed. -->
<!ENTITY % money "(#PCDATA)">

<!ELEMENT Amount %money;>
```

General entities should also be commented. Sometimes if there are a lot of them, they're all related, and the names are descriptive enough, you can settle for a single comment for a group of entities. For example, this batch declares several currency symbols.

```
<!-- Currency symbols used in statements -->
<!ENTITY YenSign   "&#0xA5;">
<!ENTITY PoundSign "&#0xA3;">
<!ENTITY EuroSign  "&#0x20AC;">
```

On the other hand, sometimes an entity is unique and needs its own comment. For example, this external general entity loads the bank's current privacy policy from a remote URL.

```
<!-- The privacy policy printed on each statement is prepared
     by the legal department and updated from time to time.
     The following URL has the most recent version at any
     given time. -->
```

```
<!ENTITY privacy SYSTEM
        "http://legal.megabank.com/privacy.xml">
```

Comments can also be used and generally should be used for declarations of elements with complex types. For example, consider the following declaration of a `transaction` element.

```
<!-- A transaction element represents any action that moves
     money into or out of an account, including, but not
     limited to:

     * A withdrawal
     * A deposit
     * A transfer
     * A loan payment
     * A fee
     * Interest credit

-->
<!ELEMENT Transaction (Account?, Date, Amount)>
```

The comment gives a much clearer picture of just what a `transaction` element is for than the mere fact that each transaction has an optional account, followed by one date and one amount.

DTDs can be quite opaque and obscure, perhaps not as obfuscated as poorly written C++ but certainly worse than clean Python code. Even when the element and attribute names are properly verbose and semantic, it's always helpful to include comments clarifying the many things the declarations themselves can't say. Remember, in some cases the DTD isn't even used for direct validation. Its primary purpose is documentation. But even when a DTD is used for validation, its use as documentation is still very important. Proper commenting makes DTDs much more effective documentation for an application.

Item 6 Name Elements with Camel Case

There are no standard naming conventions for XML. I've seen XML applications that use all capitals, all small letters, separate words with hyphens, separate words with underscores, and more. Without being too fanatical about it, I recommend camel case, usingInternalCapitalization

InLieuOfWhiteSpaceLikeThis, simply because our eyes are better trained to follow it.

In the programming sections of Usenet, case conventions are second only to indentation as a source of pointless erudition and time-wasting flameage. There are many good naming and case conventions, most of which have nothing to strongly recommend them over any other. Most modern languages like Java, Delphi, and C# have tended to adopt one convention, even if they don't enforce it in the compiler, in order to facilitate the legibility of code between different people and groups. It doesn't matter which convention is picked as long as a single convention is chosen.

XML, unfortunately, does not have any recommended naming or case conventions for element and attribute names. Multiple conventions such as those listed below are used in practice.

- XSLT uses all lower case with hyphens separating words, as in `xsl:value-of`, `xsl:apply-templates`, `xsl:for-each`, and `xsl:attribute-set`.
- The W3C XML Schema Language uses camel case with an initial lowercase letter, as in `xsd:complexType`, `xsd:simpleType`, `xsd:gMonthDay`, and `xsi:schemaLocation`.
- DocBook uses lower case exclusively and never separates the words, as in `para`, `firstname`, `biblioentry`, `chapterinfo`, `methodsynopsis`, and `listitem`.
- MathML uses lower case exclusively, does not separate the words, and furthermore often abbreviates the words, as in `mi` (math italic), `mn` (math number), `mfrac` (math fraction), `msqrt` (math square root), and `reln` (relation).
- SOAP 1.2 uses camel case with initial capitals for element names (`env:Body`, `env:Envelope`, `env:Header`, and so on) and camel case with an initial lowercase letter for attributes (`encodingStyle`, `env:role`, `env:mustUnderstand`, and so on).
- XHTML uses lower case exclusively with fairly short, abbreviated names, as in `p`, `div`, `head`, `h1`, `tr`, `td`, `img`, and so on.

The one style you tend not to see is exclusively upper case like this:

```
<STATEMENT xmlns="http://namespaces.megabank.com/">
  <BANK>MegaBank</BANK>
  <ACCOUNT>
```

```
    <NUMBER>00003145298</NUMBER>
    <TYPE>Savings</TYPE>
    <OWNER>John Doe</OWNER>
  </ACCOUNT >
  <DATE>2003-30-02</DATE>
  <OPENINGBALANCE>5266.34</OPENINGBALANCE>
  <CLOSINGBALANCE>5266.34</CLOSINGBALANCE>
</STATEMENT>
```

The reason is simple: In English and other case-aware languages (not, for example, Chinese and Hebrew) human eyes are trained to recognize words by their shape. After the first or second grade, readers do not sound out each letter when trying to identify a word. At least with common words like *first, case,* and *language,* we recognize the entire word as a unit. The ascenders and descenders of letters like *d, g, l, k,* and *j* are major contributors to the overall shape of a word. HOWEVER, IN UPPER CASE ALL LETTERS HAVE EXACTLY THE SAME HEIGHT, SO A SENTENCE WRITTEN IN PURE UPPER CASE IS MUCH HARDER TO READ. See what I mean? Consequently, a document will be much easier to read if you stick to lower case or a mix of upper and lower case.

XML vocabularies do tend to be verbose. Most vocabularies prefer to spell out the complete names of things rather than abbreviating. A few applications, such as DocBook and XHTML (in which a significant fraction of the user base authors by hand), use at least some abbreviations, but in general it's considered good form to spell out all words completely. This naturally raises the question of how to break the words. Spaces aren't legal in XML, but the next best thing is an underscore, as shown below.

```
<Opening_Balance>5266.34</Opening_Balance>
<Closing_Balance>5266.34</Closing_Balance>
```

Surprisingly, this tends not to be very common. The hyphen is a little more common but is eschewed because many data binding APIs can't easily map names containing hyphens to class names. The two most common conventions are camel case and pure lower case. However, pure lower case creates excessively long new words that are not easily recognized by their shape since they're unfamiliar. Camel case is much closer to what readers subconsciously expect. It is cleaner, easier to follow, and easier to debug. Thus I recommend using camel case.

Naturally, this recommendation does vary a little by language. If you're writing your markup in a language like Hebrew or Chinese that does not

distinguish upper and lower case, you can pretty much ignore this entire item. If you're marking up in a language like German where the nouns are distinguished by capitalization, you might choose to capitalize only the nouns. However, in English and many other languages, camel case is the most appropriate choice.

I do not have a strong opinion about whether the first letter of a camel-cased element or attribute name should be lower or upper case. As a Java programmer, I'm accustomed to seeing class names begin with uppercase letters and field names begin with lowercase letters. Probably because of that, an initial uppercase letter for elements and an initial lowercase letter for attributes seems more correct to me, but I freely admit that I can't rationalize that feeling. C# and Delphi programmers often have the opposite preference. All I can really recommend is that you pick one convention and stick with it.

Item 7 Parameterize DTDs

No one XML application can serve all uses. No one DTD can describe every necessary document. As obvious as this statement seems, there have been a number of failed efforts to develop DTDs that describe all possible documents in a given field, even fields as large as all business documents. A much more sensible approach is to design DTDs so they can be customized for different local environments. Elements and attributes can be added to or removed from particular systems. Names can be translated into the local language. Even content models can be adjusted to suit the local needs.

You cannot override attribute lists or element declarations in a DTD. However, you can override entity definitions, and this is the key to making DTDs extensible. If the attribute lists and element declarations are defined by reference to parameter entities, redefining the parameter entity effectively changes all the element declarations and attribute lists based on that parameter entity. When writing a parameterized DTD, almost everything of interest is defined as a parameter entity reference, including:

- Element names
- Attribute names
- Element content models

- Attribute types
- Attribute lists
- Namespace URIs
- Namespace prefixes

The extra level of indirection allows almost any aspect of the DTD to be changed. For example, consider a simple XML application that describes bank statements. A traditional monolithic DTD for this application might look like the listing below.

```
<!ELEMENT Statement (Bank, Account, Date, OpeningBalance,
                    Transaction*, ClosingBalance)>
<!ATTLIST Statement
    xmlns CDATA #FIXED "http://namespaces.megabank.com/">

<!ELEMENT Account (Number, Type, Owner)>
<!ELEMENT Number         (#PCDATA)>
<!ELEMENT Type           (#PCDATA)>
<!ELEMENT Owner          (#PCDATA)>
<!ELEMENT OpeningBalance (#PCDATA)>
<!ELEMENT ClosingBalance (#PCDATA)>
<!ELEMENT Bank           (#PCDATA)>
<!ELEMENT Date           (#PCDATA)>
<!ELEMENT Amount         (#PCDATA)>
<!ELEMENT Transaction (Account?, Date, Amount)>
<!ATTLIST Transaction
    type (withdrawal | deposit | transfer) #REQUIRED>
```

(I've stripped out most of the comments to save space. You can see them in Item 5 though. Of course, this example is a lot simpler than any real bank statement application would be.)

We can parameterize the DTD by defining each of the element and attribute names, content models, and types as parameter entity references. For example, the `Number` element could be defined like this:

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

However, you normally shouldn't parameterize just the name. You should also parameterize the content model.

```
<!ENTITY % Number.content " #PCDATA ">
<!ELEMENT %Number; (%Number.content;)>
```

This is longer and less clear, but it is much more extensible. For example, suppose that in a particular special document you want to change the content model from simple PCDATA to a branch code followed by a customer code like this:

```
<Number>
  <BranchCode>00003</BranchCode>
  <CustomerCode>145298</CustomerCode>
</Number>
```

You can do this in the internal DTD subset by overriding the `Number.content` parameter entity reference.

```
<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "modular_statement.dtd" [
  <!ENTITY % Number.content " BranchCode, CustomerCode ">
  <!ELEMENT BranchCode   (#PCDATA)>
  <!ELEMENT CustomerCode (#PCDATA)>
]>
```

Of course you can also do this in other DTDs that import the original DTD as well as in the internal DTD subset. You just need to be careful that the new entity definitions appear before the original entity definitions. (Declarations in the internal DTD subset are considered to come before everything in the external DTD subset.)

The same principles apply for elements such as `Transaction` with more complicated content models. Every element name is replaced by a parameter entity reference that points to the name. Every content model is replaced by a parameter entity reference that points to the content model. For example, the complete parameterized declaration of `Transaction` might look like this:

```
<!ENTITY % AccountElement     "Account">
<!ENTITY % DateElement        "Date">
<!ENTITY % AmountElement      "Amount">
<!ENTITY % TransactionElement "Transaction">
<!ENTITY % TransactionContent
            "%AccountElement;?, %DateElement;,
%AmountElement;">
<!ELEMENT %TransactionElement; ( %TransactionContent; )>
```

One of the most common adjustments to content models is adding an extra child element. While this can be done by redefining the complete content entity, it's even better to prepare for this in advance by defining an empty extra entity, as shown below.

```
<!ENTITY % Transaction.extra "">
<!ENTITY % TransactionContent
  "%AccountElement;?, %DateElement;, %AmountElement;
   %Transaction.extra;">
```

By default this adds nothing to the content model. However, redefining `Transaction.extra` allows new elements to be added.

```
<!ELEMENT ApprovedBy (#PCDATA)>
<!ENTITY % Transaction.extra ", ApprovedBy">
```

This works even better with choices than with sequences thanks to the insignificance of order. With a sequence we can add the new elements only at the end. With a choice, the new elements can appear anywhere.

Parameterizing Attributes

Attribute declarations also benefit from parameterization. For example, the `type` attribute of the `Transaction` element can be declared as follows.

```
<!ENTITY % TypeAtt "type">
<!ENTITY % type.extra "">
<!ATTLIST %TransactionElement; %TypeAtt;
  (withdrawal | deposit | transfer %type.extra;)
  #REQUIRED>
```

Now you can change the name of the attribute by redefining the `TypeAtt` entity or add an additional value by redefining the `type.extra` entity. For example, the redefinition below adds a `balanceInquiry` type.

```
<!ENTITY % type.extra " |  balanceInquiry ">
```

Parameterizing Namespaces

One of the biggest benefits of parameterization is that it lets you vary the namespace prefix. The fundamental principle of namespaces is that the

prefix doesn't matter, only the URI matters. For instance, `Statement`, `mb:Statement`, and `bank:Statement` are all the same just as long as `Statement` is in the default namespace http://namespaces.megabank.com/ and `mb:Statement` and `bank:Statement` are in contexts where the `mb` and `bank` prefixes are mapped to the http://namespaces.megabank.com/ URI. Only the local name and URI matter. The prefix (or lack thereof) doesn't.

However, DTDs violate this principle. In a DTD only the qualified name matters. The parser validates against the prefix and local name. It ignores the namespace URI. You cannot change the prefix without changing the DTD to match. However, parameterization provides a way out of this trap. First, define the prefix and the colon as separate parameter entity references.

```
<!ENTITY % statement.prefix "stmt">
<!ENTITY % statement.colon  ":">
```

Next, define the names using these parameter entities.

```
<!ENTITY % Date.qname "%statement.prefix;%statement.colon;Date">
<!ENTITY % Transaction.qname
             "%statement.prefix;%statement.colon;Transaction">
<!ENTITY % Amount.qname
             "%statement.prefix;%statement.colon;Amount">
<!ENTITY % Account.qname
             "%statement.prefix;%statement.colon;Account">

<!ENTITY % TransactionContent
             "%Account.qname;?, %Date.qname;, %Amount.qname;">
```

You'll normally also want to parameterize the namespace declaration using either an `xmlns` or `xmlns:prefix` attribute, typically on the root element. For example:

```
<!ENTITY % NamespaceDeclaration
             "xmlns%statement.colon;%statement.prefix;">
<!ATTLIST %statement.prefix;%statement.colon;%Statement.name;
  %NamespaceDeclaration; CDATA
  #FIXED "http://namespaces.megabank.com/">
```

To adjust the prefix used in a particular instance document just override `statement.prefix` and `statement.colon` in the internal DTD subset

to match what appears in the instance document. For example, the following code sets the prefix to `bank`.

```
<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "modular_statement.dtd" [
  <!ENTITY % statement.prefix "bank">
  <!ENTITY % statement.colon  ":">
]>
```

To use the default namespace set both `statement.prefix` and `statement.colon` to the empty string.

```
<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "modular_statement.dtd" [
  <!ENTITY % statement.prefix "">
  <!ENTITY % statement.colon  "">

]>
```

Warning *When parameterizing namespace prefixes like this, it is essential that you do not skip any steps. In particular, you must use the double indirection of defining entity references for both the element's qualified name and for its local name and prefix. Do not try something like this:*

```
<!ELEMENT %statement.prefix;%statement.colon;Number
                                          (%Number.content;)>
```

For various technical reasons, this will fail. In brief, without the double indirection the parser adds extra space around the resolved entities so that the resolved declaration is malformed.

Full Parameterization

After full parameterization, the entire DTD resembles the listing below.

```
<!ENTITY % statement.prefix "stmt">
<!ENTITY % statement.colon  ":">

<!ENTITY % NamespaceDeclaration
               "xmlns%statement.colon;%statement.prefix;">
```

```
<!ENTITY % Statement.qname
                "%statement.prefix;%statement.colon;Statement">
<!ENTITY % Bank.qname
                "%statement.prefix;%statement.colon;Bank">
<!ENTITY % Date.qname
                "%statement.prefix;%statement.colon;Date">
<!ENTITY % Transaction.qname
             "%statement.prefix;%statement.colon;Transaction">
<!ENTITY % Amount.qname
             "%statement.prefix;%statement.colon;Amount">
<!ENTITY % Account.qname
             "%statement.prefix;%statement.colon;Account">
<!ENTITY % Number.qname
             "%statement.prefix;%statement.colon;Number">
<!ENTITY % Owner.qname
             "%statement.prefix;%statement.colon;Owner">
<!ENTITY % Type.qname
             "%statement.prefix;%statement.colon;Type">
<!ENTITY % OpeningBalance.qname
             "%statement.prefix;%statement.colon;OpeningBalance">
<!ENTITY % ClosingBalance.qname
             "%statement.prefix;%statement.colon;ClosingBalance">

<!ATTLIST %Statement.qname; %NamespaceDeclaration;
     CDATA #FIXED "http://namespaces.megabank.com/">

<!ELEMENT %Statement.qname; (
  %Bank.qname;,
  %Account.qname;,
  %Date.qname;,
  %OpeningBalance.qname;,
  (%Transaction.qname;)*,
  %ClosingBalance.qname;)
>

<!ELEMENT Account (%Number.qname;, %Type.qname;, %Owner.qname;)>

<!ELEMENT %Number.qname; (#PCDATA)>
<!ELEMENT %Type.qname; (#PCDATA)>
<!ELEMENT %Owner.qname; (#PCDATA)>
```

```
<!ELEMENT %OpeningBalance.qname; (#PCDATA)>
<!ELEMENT %ClosingBalance.qname; (#PCDATA)>
<!ELEMENT %Bank.qname; (#PCDATA)>
<!ELEMENT %Date.qname; (#PCDATA)>
<!ELEMENT %Amount.qname; (#PCDATA)>

<!ENTITY % TransactionContent
     "%Account.qname;?, %Date.qname;, %Amount.qname;">
<!ELEMENT %Transaction.qname; ( %TransactionContent; )>

<!ENTITY % TypeAtt    "type">
<!ENTITY % type.extra "">
<!ATTLIST %Transaction.qname; %TypeAtt;
  (withdrawal | deposit | transfer %type.extra; )
  #REQUIRED
```

The downside is that this DTD is a lot less legible than the unparameterized DTD. However, the fully parameterized DTD is also much more extensible.

There are limits to all this. Sometimes you don't want to allow a DTD to be easily customized, or you want to allow it to be customized in some ways but not in others—for example, allowing elements to be added but not removed or reordered but not renamed. Generally, this is possible using the right combination of parameter entity references. I've perhaps overparameterized here to make a point, but you're free to pick and choose only those pieces that are helpful in your systems.

Conditional Sections

As the next step, we can allow particular documents to enable or disable particular parts of the DTD. This is accomplished with INCLUDE and IGNORE sections. The basic syntax for these directives appears below.

```
<![INCLUDE[
  <!-- Declarations the parser reads -->
]]>
<![IGNORE[
  <!-- Declarations the parser ignores -->
]]>
```

Note that the syntax is the same except for the keyword. By defining the keyword as a parameter entity reference, you can provide a switch customizers can use to turn sections of the DTD on or off. These can be individual declarations or groups of related declarations.

For example, some bank subsidiaries with different systems might wish to leave out an explicit closing balance since it can be calculated from the opening balance and the individual transactions. To enable this, you'd first define a parameter entity such as this one:

```
<!ENTITY % IncludeClosingBalance "INCLUDE">
```

Next, you'd place the declaration of the `ClosingBalance` element inside a conditional section that can be either included or ignored depending on the value of the `IncludeClosingBalance` entity.

```
<![%IncludeClosingBalance;[
  <!ELEMENT %ClosingBalance.qname; (#PCDATA)>
]]>
```

What about the content models that use `ClosingBalance`? How can they be parameterized? This is quite tricky, but it can be done. You'll need two declarations, one for the case that includes the closing balance and one for the case that doesn't. You'll need to define two parameter entities that identify the two cases. Do this both inside and after the conditional block that decides whether or not to use closing balances.

```
<![%IncludeClosingBalance;[
  <!ELEMENT %ClosingBalance.qname; (#PCDATA)>
  <!ENTITY  AddClosingBalance     "INCLUDE">
  <!ENTITY  DontAddClosingBalance "IGNORE">
]]>
<!ENTITY  AddClosingBalance     "IGNORE">
<!ENTITY  DontAddClosingBalance "INCLUDE">
```

Inside the conditional block the entities have the value to be used if the closing balance is included. After the conditional block the entities have the value to be used if the closing balance is not included. Order matters here. We're relying on the fact that the first declaration the parser encounters takes precedence when picking between the two possibilities.

Next, wrap the two different declarations of the `Statement` element in their own conditional sections based on the `AddClosingBalance` and `DontAddClosingBalance` entities.

```
<![%AddClosingBalance;[
  <!ELEMENT %Statement.qname; (
  %Bank.qname;,
  %Account.qname;,
  %Date.qname;,
  %OpeningBalance.qname;,
  (%Transaction.qname;)*,
  %ClosingBalance.qname;)
>
]]>
<![%DontAddClosingBalance;[
<!ELEMENT %Statement.qname; (
  %Bank.qname;,
  %Account.qname;,
  %Date.qname;,
  %OpeningBalance.qname;,
  (%Transaction.qname;)*)
>
]]>
```

I don't recommend going this far for all declarations in all DTDs, but when you need this much flexibility, this approach is very convenient.

Parameterization recognizes the reality that one size does not fit all. Different systems need to supply and support different information. Making DTDs parameterizable enables them to be customized and reused in ways never envisioned by the original developers. At a minimum you should allow for parameterization that does not remove information from the document: changing namespace prefixes and adding attributes and content to elements. Allowing the complete redefinition of element content models is perhaps a little more dangerous. It has the potential to break processes that depend on the information guaranteed to be provided by a document valid against the unparameterized DTD. However, even this can be useful as long as application software takes proper care not to assume too much and the customized DTD uses a different public ID so it's easily distinguished from the uncustomized variant. Parameterization is not always necessary for tightly coupled systems inside one organization, but its enhanced flexibility is very important for the loosely coupled systems of the Internet, where widely dispersed organizations routinely use applications for tasks the original developers never imagined.

Item 8 Modularize DTDs

Large, monolithic DTDs are as hard to read and understand as large, monolithic programs. While DTDs are rarely as large as even medium-sized programs, they can nonetheless benefit from being divided into separate modules, one to a file. Furthermore, this allows you to combine different DTDs in a single application. For example, modularization allows you to include your own custom vocabularies in XHTML documents.

Modularization divides a DTD into multiple, somewhat independent units of functionality that can be mixed and matched to suit. A master DTD integrates all the parts into a single driver application. Parameterization is based on internal parameter entities. Modularization is based on external parameter entities. However, the techniques are much the same. By redefining various entities, you choose which modules to include where.

I'll demonstrate with another variation of the hypothetical MegaBank statement application used in earlier items. This time we'll look at a statement that's composed of several more or less independent parts. It begins with a batch of information about the bank branch where the account is held, then the list of transactions, and finally a series of legal fine print covering things like the bank's privacy policy and where to write if there's a problem with the statement. This latter part is written in XHTML. Example 8–1 is a complete bank statement that demonstrates all the relevant parts.

Example 8–1 | A Full Bank Statement

```
<?xml version="1.0"?>
<!DOCTYPE Statement PUBLIC "-//MegaBank//DTD Statement//EN"
                           "statement.dtd">
<Statement xmlns="http://namespaces.megabank.com/">
  <Bank>
    <Logo href="logo.jpg" height="125" width="125"/>
    <Name>MegaBank</Name>
    <Motto>We Really Pretend to Care</Motto>
    <Branch>
       <Address>
          <Street>666 Fifth Ave.</Street>
```

```
            <City>New York</City>
            <State>NY</State>
            <PostalCode>10010</PostalCode>
            <Country>USA</Country>
        </Address>
    </Branch>
</Bank>
<Account>
    <Number>00003145298</Number>
    <Type>Savings</Type>
    <Owner>John Doe</Owner>
    <Address>
        <Street>123 Peon Way</Street>
        <Apt>28Z</Apt>
        <City>Brooklyn</City>
        <State>NY</State>
        <PostalCode>11239</PostalCode>
        <Country>USA</Country>
    </Address>
</Account>
<Date>2003-30-02</Date>
<AccountActivity>
    <OpeningBalance>5266.34</OpeningBalance>
    <Transaction type="deposit">
        <Date>2003-02-07</Date>
        <Amount>300.00</Amount>
    </Transaction>
    <Transaction type="transfer">
        <Account>
            <Number>0000271828</Number>
            <Type>Checking</Type>
            <Owner>John Doe</Owner>
        </Account>
        <Date>2003-02-07</Date>
        <Amount>200.00</Amount>
    </Transaction>
    <Transaction type="deposit">
        <Date>2003-02-15</Date>
        <Amount>512.32</Amount>
    </Transaction>
```

```
    <Transaction type="withdrawal">
      <Date>2003-02-15</Date>
      <Amount>200.00</Amount>
    </Transaction>
    <Transaction type="withdrawal">
      <Date>2003-02-25</Date>
      <Amount>200.00</Amount>
    </Transaction>
    <ClosingBalance>5488.66</ClosingBalance>
</AccountActivity>
<Legal>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <body style="font-size: xx-small">
       <h1>Important Information About This Statement</h1>
       <p>
         In the event of an error in this statement,
         please submit complete details in triplicate to:
       </p>

       <p>
         MegaBank Claims Department
         3 Friday's Road
         Adamstown
         Pitcairn Island
       </p>

       <p>
         We'll get back to you in four to six weeks.
        (We're not sure which four to six weeks, but we do
         know it's bound to be some period of four to six
         weeks, sometime.)
       </p>

       <h2>Privacy Notice</h2>
       <p>
         You have none. Get used to it. We will sell your
         information to the highest bidder, the lowest
         bidder, and everyone in between. We'll give it
         away to anybody who can afford a self-addressed
         stamped envelope. We'll even trade it for a
         lifetime supply of Skinny Dip Thigh Cream & trade;.
```

```
            It's not like it costs us anything.
          </p>
        </body>
      </html>
    </Legal>
</Statement>
```

As usual, a real-world document would be considerably more complex and contain a lot more data, but this is enough to present the basic ideas.

Leaving aside comments, a modular DTD normally starts in a driver module. This is a DTD fragment that declares a couple of crucial entity references, such as those defining the namespace URI and prefix, then loads the other modules. For example, in the bank statement application, a driver DTD might look something like Example 8–2.

Example 8–2 | The Statement DTD Driver Module

```
<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % stmt.prefix "" >

<!ENTITY % stmnt-qnames.mod SYSTEM "stmnt-qnames.mod" >

<!ENTITY % stmnt-framework.mod SYSTEM "stmnt-framework.mod" >
%stmnt-framework.mod;

<!ENTITY % stmnt-structure.mod SYSTEM "stmnt-structure.mod" >
%stmnt-structure.mod;

<!ENTITY % stmnt-address.mod SYSTEM "stmnt-address.mod" >
%stmnt-address.mod;

<!ENTITY % stmnt-branch.mod SYSTEM "stmnt-branch.mod" >
%stmnt-branch.mod;

<!ENTITY % stmnt-transaction.mod SYSTEM "stmnt-transaction.mod" >
%stmnt-transaction.mod;

<!ENTITY % stmnt-legal.mod SYSTEM "stmnt-legal.mod" >
%stmnt-legal.mod;
```

Since all the modules are loaded via entity references, this allows local bank branches and subsidiaries to substitute their own modules by redefining those entities or by writing a different driver.

In this case, the various parts have been loaded in the following order.

1. Specify whether namespace prefixes are used. (Here they aren't, by default.)
2. Define the location of the qualified names module. This will be loaded by the framework.
3. Define and load the framework, which is responsible for loading common DTD parts that cross module boundaries, such as the qualified names module, the entities module, and any common content models or attribute definitions shared by multiple elements.
4. Define and load the structure module, which is responsible for merging the different modules into a single DTD for a complete document.
5. Define the separate modules that comprise different, somewhat independent parts of the application. Here there are four:
   a. The address module
   b. The branch module
   c. The transaction module
   d. The legal module

Conditional sections can control which modules are or are not included. This makes the DTD a little harder to read, which is why I didn't show it this way in the first place; but they're essential for customizability. Example 8–3 demonstrates.

Example 8–3 | The Conditionalized Statement DTD Driver Module

```
<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % stmt.prefix "" >

<!-- Address Module   -->
<!ENTITY % stmnt-address.module "INCLUDE" >
<![%stmnt-address.module;[
<!ENTITY % stmnt-address.mod
     SYSTEM "stmnt-address.mod" >
%stmnt-address.mod;]]>

<!ENTITY % stmnt-branch.module "INCLUDE" >
<![%stmnt-branch.module;[
<!ENTITY % stmnt-branch.mod
     SYSTEM "stmnt-branch.mod" >
%stmnt-branch.mod;]]>
```

```
<!ENTITY % stmnt-qnames.mod
     SYSTEM "stmnt-qnames.mod" >

<!ENTITY % stmnt-framework.mod SYSTEM "stmnt-framework.mod" >
%stmnt-framework.mod;

<!ENTITY % stmnt-transaction.module "INCLUDE" >
<![%stmnt-transaction.module;[
<!ENTITY % stmnt-transaction.mod
     SYSTEM "stmnt-transaction.mod" >
%stmnt-transaction.mod;]]>

<!ENTITY % stmnt-legal.module "INCLUDE" >
<![%stmnt-legal.module;[
<!ENTITY % stmnt-legal.mod SYSTEM "stmnt-legal.mod" >
%stmnt-legal.mod;]]>
```

Not all pieces can be turned on or off. Generally, the framework and the qualified names modules are required and thus are not wrapped in conditional sections.

Now let's take a look at what you might find inside the individual modules. The qualified names module (Example 8–4), which has not yet been loaded, only referenced, defines the names of elements that will be used in different content models using the parameterization techniques shown in Item 7.

Example 8–4 | The Qualified Names Module

```
<!ENTITY % Statement.qname
     "%statement.prefix;%statement.colon;Statement">
<!ENTITY % Bank.qname
     "%statement.prefix;%statement.colon;Bank">
<!ENTITY % Date.qname
     "%statement.prefix;%statement.colon;Date">
<!ENTITY % Transaction.qname
     "%statement.prefix;%statement.colon;Transaction">
<!ENTITY % Amount.qname
     "%statement.prefix;%statement.colon;Amount">
<!ENTITY % Account.qname
     "%statement.prefix;%statement.colon;Account">
<!ENTITY % Number.qname
```

```
     "%statement.prefix;%statement.colon;Number">
<!ENTITY % Owner.qname
     "%statement.prefix;%statement.colon;Owner">
<!ENTITY % Type.qname
     "%statement.prefix;%statement.colon;Type">
<!ENTITY % OpeningBalance.qname
     "%statement.prefix;%statement.colon;OpeningBalance">
<!ENTITY % ClosingBalance.qname
     "%statement.prefix;%statement.colon;ClosingBalance">

<!ENTITY % Logo.qname "%statement.prefix;%statement.colon;Logo">
<!ENTITY % Name.qname "%statement.prefix;%statement.colon;Name">
<!ENTITY % Motto.qname
    "%statement.prefix;%statement.colon;Motto">
<!ENTITY % Branch.qname
    "%statement.prefix;%statement.colon;Branch">
<!ENTITY % Address.qname
    "%statement.prefix;%statement.colon;Address">
<!ENTITY % Street.qname
    "%statement.prefix;%statement.colon;Street">
<!ENTITY % Apt.qname "%statement.prefix;%statement.colon;Apt">
<!ENTITY % City.qname "%statement.prefix;%statement.colon;City">
<!ENTITY % State.qname
    "%statement.prefix;%statement.colon;State">
<!ENTITY % PostalCode.qname
    "%statement.prefix;%statement.colon;PostalCode">
<!ENTITY % Country.qname
    "%statement.prefix;%statement.colon;Country">
<!ENTITY % AccountActivity.qname
    "%statement.prefix;%statement.colon;AccountActivity">
<!ENTITY % Legal.qname
    "%statement.prefix;%statement.colon;Legal">
```

The framework module shown in Example 8–5 actually loads the qualified names module. In addition, it loads any general modules used across the DTD, such as those for defining common attributes or character entities. In other words, the framework module defines those modules that cross other module boundaries.

Example 8–5 | The Framework Module

```
<!ENTITY % stmnt-qname.mod SYSTEM "stmnt-qnames.mod" >
%stmnt-qname.mod;

<!ENTITY % stmnt-attribs.module "INCLUDE" >
<![%stmnt-attribs.module;[
<!ENTITY % stmnt-attribs.mod SYSTEM "stmnt-attribs.mod" >
%stmnt-attribs.mod;
]]>

<!ENTITY % stmnt-model.module "INCLUDE" >
<![%stmnt-model.module;[%stmnt-model.mod;
]]>

<!ENTITY % stmnt-charent.module "INCLUDE" >
<![%stmnt-charent.module;[
<!ENTITY % stmnt-charent.mod SYSTEM "stmnt-charent.mod" >
%stmnt-charent.mod;
]]>
```

The structure module (Example 8–6) defines the root element. It provides the overall architecture of a statement document, that is, a `Statement` element that contains a variety of child elements mostly drawn from other modules.

Example 8–6 | The Structure Module

```
<!ELEMENT %Statement.qname; (
  %Bank.qname;,
  %Account.qname;,
  %Date.qname;,
  %AccountActivity.qname;,
  %Legal.qname;
)>

<!ENTITY % NamespaceDeclaration
    "xmlns%statement.colon;%statement.prefix;">

<!ATTLIST %Statement.qname; %NamespaceDeclaration;
              CDATA #FIXED "http://namespaces.megabank.com/">
```

The final step is to define the individual modules for the different classes of content. Example 8–7 demonstrates the transaction module. It's very similar to the parameterized version of the DTD shown in Item 7.

Example 8–7 | The Transaction Module

```
<!ELEMENT %AccountActivity.qname; (
    %OpeningBalance.qname;,
    (%Transaction.qname;)*,
    %ClosingBalance.qname;
)>

<!ENTITY  % OpeningBalance.content " #PCDATA ">
<!ELEMENT %OpeningBalance.qname; (%OpeningBalance.content;)>
<!ENTITY  % ClosingBalance.content " #PCDATA ">
<!ELEMENT %ClosingBalance.qname; (%ClosingBalance.content;)>
<!ENTITY  % Amount.content " #PCDATA ">
<!ELEMENT %Amount.qname; (%Amount.content;)>
<!ENTITY  % Date.content " #PCDATA ">
<!ELEMENT %Date.qname; (%Date.content;)>

<!ENTITY  % TransactionContent
    "%Account.qname;?, %Date.qname;, %Amount.qname;">
<!ELEMENT %Transaction.qname; ( %TransactionContent; )>

<!ENTITY % TypeAtt    "type">
<!ENTITY % type.extra "">
<!ATTLIST %Transaction.qname; %TypeAtt;
  (withdrawal | deposit | transfer %type.extra; )
  #REQUIRED
>
```

Notice how almost everything has been parameterized with entity references so that almost any piece can be changed independent of the other pieces. The other modules are similar. In general, the dependencies are limited and unidirectional. The modules depend on the framework and the document model, but not vice versa. This allows you to add and remove modules by adjusting the document model and the framework alone. You can change the individual parts of the module by redefining the various entities. Together with parameterization this makes the DTD extremely flexible. Not all DTDs require this level of customizability, but for those that do, modularization is extremely powerful.

Item 9 Distinguish Text from Markup

All legal text characters that can appear anywhere in an XML document can appear in #PCDATA. This includes characters like < and & that may have to be escaped with character or entity references. When an API presents the content of a node containing such a character to your code, it will give you the actual character, not the escaping text. Similarly, when you create such a node, the string you use should contain the actual character, not the entity or character reference.

Consider the following DocBook `programlisting` element. A CDATA section is used to embed a literal sequence of XML text.

```
<programlisting><![CDATA[<value>
  <double>28657</double>
 </value>]]></programlisting>
```

Everything inside the CDATA section is content, not markup. The content of this `programlisting` element is the text shown below.

```
<value>
  <double>28657</double>
</value>
```

A CDATA section is not required for this trick to work. For instance, consider the following variation of the above element.

```
<programlisting>&lt;value&gt;
  &lt;double&gt;28657&lt;/double&gt;
 &lt;/value&gt;</programlisting>
```

The content of this element is exactly the same.

```
<value>
  <double>28657</double>
</value>
```

In this case the markup of the entity references `<` and `>` is resolved to produce the text < and >. However, that's just syntax sugar. It does not affect the content in any way.

Now consider the reverse problem. Suppose you're creating an XML document in something at least a little more XML-aware than a text editor. Possibilities include:

- A tree-based editor like <Oxygen/> or XMLSPY
- A WYSIWYG application like OpenOffice Writer or Apple's Keynote that saves its data into XML
- A programming API such as DOM, JDOM, or XOM

In all cases, the creating tool will provide separate means to insert markup and text. The tool is responsible for escaping any reserved characters like <, >, or & when it saves the document. You do not need to do this. Indeed, if you try to pass something like `<double>28657 </double>` into a method that expects to get plain text, it will actually save something like `&lt;double&gt;28657 &lt;/double&gt;`.

Similarly, you cannot type `<double>28657</double>` into a user interface widget that creates text and expect it to create an element. If you try it, in the serialized document you will get something like `<double>28657</double>`. Instead, you should use the user interface widget or method call designed for creating a new element.

The key thing to remember is this: Just because something looks like an XML tag does not always mean it *is* an XML tag. Context matters. XML documents are made of markup that sometimes surrounds PCDATA, but that's the limit of the nesting. You can put PCDATA inside markup, and you can put markup inside markup, but you can't put markup inside PCDATA. CDATA sections are just an alternative means of escaping text. They are not a way to embed markup inside PCDATA.

Item 10 White Space Matters

XML defines white space as the Unicode characters space (0x20), carriage return (0x0D), line feed (0x0A), and tab (0x09), as well as any combination of them. Other invisible characters, such as the byte order mark and the nonbreaking space (0xA0), are treated the same as visible characters, such as A and $.

White space is significant in XML character data. This can be a little surprising to programmers who are used to languages like Java where white space mostly isn't significant. However, remember that XML is a markup language, not a programming language. An XML document contains data, not code. The data parts of a program (that is, the string literals) are precisely where white space does matter in traditional code. Thus it really shouldn't be a huge surprise that white space is significant in XML.

For example, the following two `shape` elements are not the same.

```
<shape>star</shape>
```

```
<shape>
     star
</shape>
```

Depending on the context, a particular XML application may choose to treat these two elements the same. However, an XML parser will faithfully report all the data in both `shape` elements to the client application. If the client application chooses to trim the extra white space from the content of the second element, that's the client application's business. XML has nothing to do with it.

The `xml:space` Attribute

The `xml:space` attribute can indicate the significance (or lack thereof) of white space within a particular element. It has two legal values, `default` and `preserve`. The value `default` means that the application may treat white space in the element in whatever fashion is customary for that application. For instance, it might trim off or compress excess white space. The value `preserve` means that white space is significant, even if it normally isn't, and the application should not adjust the white space in any way.

For example, the `xml:space` attribute of the `text` element below indicates that white space is significant and should be preserved.

```
<text xml:space="preserve">
  I try to make XML lean
  Without extra white space between
  But when I forgo
  All the spaces I know
  My markup is not at all clean.
</text>
```

However, even without this attribute, the parser would *still* report all white space in the element content to the client application. It is the responsibility of the client application to inspect the value of the `xml:space` attribute and normalize the white space accordingly (or not).

Ignorable White Space

Another special case that really isn't all that special is white space in element content, sometimes misleadingly called *ignorable white space*. This is white space in an element whose element declaration specifies that it can contain only child elements and not PCDATA. For example, suppose the `PhoneNumber` element has the following declaration.

```
<!ELEMENT PhoneNumber (CountryCode, AreaCode, Number)>
```

Now consider the valid `PhoneNumber` element below.

```
<PhoneNumber>
  <CountryCode>01</CountryCode>
  <AreaCode>212</ AreaCode >
  <Number>555-1234</ Number >
</PhoneNumber>
```

Because the DTD says this element can only contain these three child elements, the white space is ignorable. It is assumed to exist only for formatting and not to have any real purpose. Nonetheless, the parser will *still* report all of it to the client application. In most APIs, there will be no distinction between this white space and any other nonignorable white space. The notable exception here is SAX. SAX will pass this white space to the `ignorableWhiteSpace()` method rather than to the `characters()` method, so the client application can distinguish ignorable from nonignorable white space.

Tags and White Space

Inside tags the white space story is quite different. These three `shape` elements are the same for all intents and purposes.

```
<shape>star</shape>

<shape
            >star</shape>

<shape    >star</shape>
```

Parsers will not distinguish one from the other, and your code should not depend on the difference.

Similarly, white space in a document's prolog, epilog, and DTD is not considered. For example, these three documents are essentially the same.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="root.css"?>
<!DOCTYPE root SYSTEM "data.dtd">
<root>
  contents
</root>
```

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

<?xml-stylesheet type="text/css" href="root.css"?>

<!DOCTYPE root SYSTEM "data.dtd">

<root>
  contents
</root>
```

```
<?xml version="1.0"?><?xml-stylesheet type="text/css"
                                        href="root.css"?><!DOCTYPE
 root SYSTEM "data.dtd">
<root>
  contents
</root>
```

White Space in Attributes

The final and trickiest case are attribute values. Depending on attribute type, the parser *normalizes* attribute values before reporting them to the client application. First it converts all tabs, carriage returns, and line feeds to one space each. This is done for all attributes. Next, if the attribute has a declared type and that type is anything other than CDATA, the parser also condenses all runs of space to a single space and finally trims all leading and trailing white space from the value. However, the parser does not perform this second step for attributes that have type CDATA or are undeclared.

For example, consider the following document. The parser will trim the leading and trailing white space from the `year` attribute because it has type NMTOKEN, but it will not trim the white space from the `source` attribute that has type CDATA or the `group` attribute that is undeclared.

```
<?xml version="1.0"?>
<!DOCTYPE motto [
  <!ATTLIST motto year   NMTOKEN #IMPLIED
                 source CDATA   #IMPLIED
]>
<motto year=" 1908 " source=" Scouting  for  Boys " group="
BSA ">
  Be prepared
</motto>
```

It is not necessary for the document to be valid in order for normalization to apply—indeed, the document above is not valid—only that the attribute be declared on the element where it appears and that the parser read that declaration. All conforming XML parsers are required to read the internal DTD subset (up to the first external parameter entity reference they don't read) and use `ATTLIST` declarations in the internal DTD subset to decide whether to normalize or not. However, if an attribute is declared in the external DTD subset, then nonvalidating parsers may or may not read the declaration. This means validating and nonvalidating parsers can report different values for the same attribute, as can two nonvalidating parsers. The values will differ only in white space, but this can still be important. If this is a concern, make sure you use either a fully validating parser or a nonvalidating parser that is known to read the external DTD subset.

Note *Tim Bray, one of the primary authors of XML 1.0, has admitted that normalization of attribute values was a mistake. In his words, "Why the $#%%!@! should attribute values be 'normalized' anyhow? This was a pure process failure: at no point during the 18-month development cycle of XML 1.0 did anyone stand up and say 'why are you doing this?' I'd bet big bucks that if someone had, the silly thing would have died a well-deserved death."*[1]

1. "Re: Attribute normalisation and character entities," posted on the xml-dev mailing list, January 27, 2000. Accessed in June 2003 at http://www.lists.ic.ac.uk/hypermail/xml-dev/xml-dev-Jan-2000/1085.html.

Schemas

The W3C XML Schema Language uses the `whiteSpace` facet to specify whether white space in an attribute or element with a simple type should be preserved, replaced, or collapsed.

- *Preserved white space* is like `xml:space="preserve"`; that is, all white space is considered to be significant.
- *Replaced white space* replaces all tabs, carriage returns, and line feeds with a single space each. Thus the number of white space characters is preserved but their type may be changed.
- *Collapsed white space* first replaces all carriage returns and line feeds with a single space each, then replaces each run of consecutive spaces with a single space, and finally trims all leading and trailing white space.

However, once again this only offers a hint to the client application. The parser will *still* report all white space to the client application in the same way it would without the schema. Indeed, the `whiteSpace` facet doesn't even change the set of valid content for an element. For example, suppose the `shape` element is declared in a schema like the one below.

```
<xsd:element name="shape">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:whiteSpace value="collapse"/>
    </xsd:restriction>
</xsd:element>
```

The following three elements are all still valid.

```
<shape> triangle    rectangle  square  </shape>

<shape>triangle rectangle square</shape>

<shape>
      triangle
      rectangle
      square
</shape>
```

Bottom line: White space is significant in XML documents except in the prolog, epilog, and tags. Parsers report all white space in element content

and attribute values to the client application. Depending on the DTD, the `xml:space` attribute, and the value of the `whiteSpace` facet in the schema, parsers may indicate that white space is or is not significant. However, it will all be reported. It is up to each individual application to determine the rules by which white space is treated.