Advanced XML

CHAPTER

4

"Readers and users be forewarned, however; if you thought Java was a rush of wave on wave of new standards, APIs, and tools—surf's up! XML is incoming."

—Jacques Surveyer, "XML Meets Java," Java Pro, April 1999

IN THIS CHAPTER

- **Advanced Markup 146**
- **Namespaces in XML 160**
- **The XLink Specification 164**
- **The XPointer Specification 168**
- **XML Schemas 172**

In Chapter 1, "An XML Primer," you learned just enough XML to begin experimenting with programming it using SAX and DOM. In this chapter, I will cover areas that we previously glossed over and advanced topics not yet discussed. Due to this "fill-in-the-blanks" approach, this chapter will sometimes skip from topic to topic with little cohesion between the topics. However, in the end, what you learn in this chapter combined with your knowledge acquired in the previous chapters will give you a thorough understanding of XML.

Advanced Markup

In this section, we will cover advanced topics of XML markup like entities, mixed and content specifications, and the remaining attribute types. We begin with how and why to add character references to your XML document.

Character References

XML uses the *Unicode* character set. Unicode is a standard that allows characters from all existing, and even ancient, languages. If your keyboard does not have a key for a particular international character you can insert that character into an XML document using a *character reference*.

Character Representation

It is important to distinguish between the varying facets of representing characters in documents. Each definition represents one facet of such representation:

- *Character*—A letter in a language.
- *Glyph*—The picture or rendered illustration of the character.
- *Coded character set*—An agreed-on mapping of characters to positions in a code space. XML uses the Unicode character set. Unicode supports a base of 1,114,112 positions and each character can be encoded in one or two 16-bit words.
- *Font*—A collection of glyphs for a character set.
- *Character set encoding*—The final step in using characters in an electronic document is deciding how to represent the numeric positions in the chosen character set in binary form in the file on disk. Simple character sets like ASCII or Latin1 encode using a byte of storage per character, with the value of the byte being the integer value of the position in the character set. This only worked because these character sets only supported 128 and 256 characters, respectively. Because Unicode is a much larger character set, some other encoding is needed. There are four possible encodings of Unicode characters: UCS-2, UTF-7, UTF-8,

and UTF-16. The last two are the most common. UCS-2 only encodes the first 65,536 positions. UTF-7 uses only the first seven bits in a byte and was suitable for older email handling agents. UTF stands for Universal Character Set Transformation Format. UTF-8 uses one or more eight-bit bytes to encode Unicode characters with the first 256 characters encoding using a single byte just like Latin1. UTF-16 is similar to UCS-2 but has an escape mechanism to encode all the Unicode characters.

There are two formats for a character reference: decimal and hex. The decimal representation is

```
CharRef :== '&#' [0-9]+ ';'
```

For example:

```
<P> Here is a special character: &#169; </P>
```

In Unicode, as in Latin1, position 169 is the copyright symbol (©).

The hex representation is

```
CharRef :== '&#x' [0-9a-fA-F]+ ';'
```

Here is an example of a character reference in hex:

```
<P> Here is another special character: &#xB6; </P>
```

In decimal, this character is position 182, which is a paragraph symbol.

NOTE

The Unicode standard lists its characters in hex notation.

Just as character references are abbreviations for Unicode characters, the XML specification also allows you to define your own abbreviations called entities and refer to them with entity references.

Entities and Entity References

Entities and entity references are techniques for enabling reuse of both content and markup in your XML documents. Entities can be confusing because there are many different categories of entities; however, after you clearly understand the various categories and to what category a particular entity belongs, they make sense. Figure 4.1 depicts the hierarchy of entity categories.

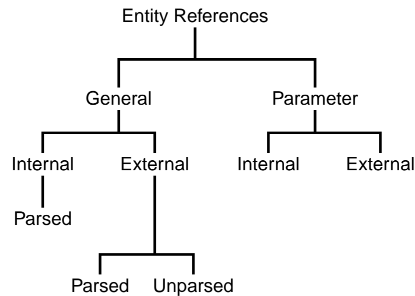

FIGURE 4.1
Hierarchy of entity categories.

Before demonstrating each category, let's look at Table 4.1, which provides definitions of each category.

TABLE 4.1 Entity Categories

Category	*Definition*
General entity	Entities used only in document content.
Parameter entity	Entities used only in document type definitions (DTDs).
Parsed entity	An entity whose replacement content is text.
Unparsed entity	Normally used for binary (non-text entities). If it is text, it may not be XML. Has an associated notation.
Internal parsed entity	An entity that is declared in an instance of the XML document.
External entity	An entity fetched from an external source. The external source is specified via a URI.

NOTE

An internal entity must be a parsed entity.

General Entities

In its simplest form (an internal, general parsed entity) an entity is just an abbreviation for larger text. For example, an entity `dtd` could be used to abbreviate the phrase *document type definition*.

You declare entities in your document type definition like this:

```
<!ENTITY dtd  "document type definition">
```

Another way to think about an entity is as a box with a label. The label is the entity's name. The contents of the box can be text or data. If the content of the entity is text, the standard calls this a parsed entity. Because the content is text it may also contain markup. For example:

```
<!ENTITY line "<P>This is a parsed entity. </P>">
```

An entity can be fetched from an external source specified by a URI. This is called an external entity. Here's another example:

```
<!ENTITY intro SYSTEM "http://www.gosynergy.com/intro.xml">
```

In an XML document, an entity is referred to via an *entity reference*. Here is the formal definition of an entity reference.

```
EntityRef :== '&'  Name  ';'
```

Table 4.2 shows the predefined entities for the characters used to delineate markup

TABLE 4.2 Predefined Entities

Entity Reference	*Character*
&	&
<	<
>	>
'	'
"	"

NOTE

Unlike HTML, which has many predefined entities, the entities listed in Table 4.2 are the only ones predefined in XML.

Here is a more complete example that uses a parsed general entity:

```
<!DOCTYPE BOOK [
<!ENTITY publisher "SAMS Professional publishing">
]>
<BOOK>
<PUBLISHER> &publisher; </PUBLISHER>
<P> Welcome to this book published by &publisher;.  This
```

```
&publisher; produces numerous professional titles every year.
</BOOK>
```

There are also unparsed entities for data such as images:

```
<!ENTITY image SYSTEM "http://www.wyweb.com/myhouse.gif" NDATA GIF>
```

Notice that unparsed entities are differentiated with the keyword `NDATA` followed by a notation name for that data. The notation name must be a declared notation. You declare notations in the DTD similar to the way you declare elements. The declaration of the GIF notation would be

```
<!NOTATION GIF SYSTEM "apps/imgviewer.exe ">
```

So far we have seen parsed and unparsed entities and external and internal entities. There is one more distinction that can be applied to entities: *general* or *parameter*. So far, we have seen only general entities. Remember, a general entity is an entity used for text replacement in a document instance.

Parameter Entities

A *parameter* entity is an entity that is only used in a DTD. It is differentiated in both its declaration and reference by a `%` symbol.

Here is an example of an internal parameter entity declaration:

```
<!DOCTYPE EXAMPLE [
<!ENTITY % obj "<!ELEMENT OBJECT (#PCDATA)>">
%obj;
]>
```

Parameter entities have different rules for an internal DTD (called an internal subset) versus an external DTD (called an external subset). In an internal DTD you can only have whole declarations (as shown above). In an external DTD you can have a parameter entity for partial declarations. This is to make parsing for non-validating parsers (which must parse the internal subset) easier—for example:

```
<!ENTITY % nameAtt  "name CDATA #REQUIRED">
<!ATTLIST folder
         %nameAtt;>
<!ATTLIST bookmark
         %nameAtt;
         type CDATA #IMPLIED>
```

External parameter entities allow you to reuse common declarations. For example, an employee element may be used in several different markup languages across the business. Here is an example of an external parameter entity:

```
<!ENTITY % employee SYSTEM "http://www.super.com/xml/employee.dtd">
...
%employee;
```

NOTE

Markup may not span entity boundaries. The following is illegal:

```
<!DOCTYPE SAMPLE [
<!ENTITY start-tag  "<title>This is very">
<!ENTITY end-tag  "illegal. </title>">
]>
&start;&finish;
```

I'd like to make the following points about entities:

- In an attribute value you can use an internal, general entity. For example:

```
<!ENTITY favrest "Tippy's Taco house">
<!ATTLIST menu
          date CDATA #REQUIRED
          restaurant CDATA #FIXED "&favrest;">
```

- Entities must be declared before they are used.

Entities are a concept that will take experience to master. A good way to speed up your learning curve on entities is to examine DTDs and documents written by others. See XML.org for a repository and catalog of XML documents and DTDs.

Understanding Attribute Types

In Chapter 2, "Parsing XML," you learned how to declare attributes in a document type declaration using an attribute-list declaration. For example:

```
<!ELEMENT CONTACT (#PCDATA)>
<!ATTLIST  CONTACT  EMAIL CDATA #REQUIRED>
```


Attributes have types that enforce both lexical and semantic constraints. Table 4.3 provides a summary of the attribute types.

TABLE 4.3 Attribute Types

Type	*Definition*
`CDATA`	Any character data
`Enumeration`	A list of `Nmtokens` ("name tokens") where only one may be used (similar to a choice content model)
`NOTATION`	A list of names and a declared notation name
`ID`	A name that uniquely identifies an element

TABLE 4.3 Continued

Type	*Definition*
`IDREF`	A reference to an element (by its `ID`)
`IDREFS`	May refer to one or more `ID`s (space delimited)
`ENTITY`	A name of a declared entity
`ENTITIES`	One or more declared entities (space delimited)
`NMTOKEN`	An `Nmtoken` (see definition later in the chapter)
`NMTOKENS`	One or more `Nmtokens`

Many of the definitions in Table 4.3 specify using either a name or a name token. A *name* is any valid XML name. An XML name must begin with a letter or an underscore, followed by any number of letters, digits, hyphens, underscores, periods, or colons. Colons are now used to denote namespaces (discussed next). XML names are used for all element and attribute names. For example:

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

An `NmToken` or *name token* is any combination of legal name characters for XML names. In other words, all XML names are name tokens, but not all name tokens have XML names. Here are some sample name tokens:

```
.1.a.name.token.but.not.a.name
234_also_a_name_token_but_not_a_name
A_name_token_and_a_name
```

In Chapter 2, I covered the most common attribute data types (`CDATA` and enumeration); now I will both define and demonstrate all the available attribute types.

- `CDATA` is the simplest type of attribute. It allows any character data except `<`, `&` (unless it starts a reference), or the quotation character used to surround the string. For example:

  ```
  <!ATTLIST QUOTE  DATE  CDATA #REQUIRED>
  ]>
  <QUOTE  DATE="February 9, 1999"> ... </QUOTE>
  ```

- An Enumeration type allows an attribute to take one name token among a choice of any number of name tokens. For example:

  ```
  <!ATTLIST CHOICE (option1|option2|option3) #REQUIRED>
  ```

- Name token (`NMTOKEN`) attributes are similar to `CDATA` except that they are restricted to valid name tokens (only name characters). An empty string is not a valid name token. Also, a name token cannot have whitespace. For example:

  ```
  <!ATTLIST QUOTE DATE NMTOKEN #REQUIRED>
  ```

```
... ]>
<QUOTE  DATE="1999-02-09"> ... </QUOTE>
```

The NMTOKENS declaration allows an attribute value to be one or more NMTOKENS separated by a space.

- An ID attribute allows you to name a particular element so that it may be referred to later using an IDREF attribute. These ID attributes will also be used with XLINKS, which are discussed later. IDs are XML names. Every element can have at most one ID. All IDs specified in an XML document must be unique. IDREF attributes must refer to an ID in the document. Also, if you use the IDREFs designation, you may have an attribute that has one or more IDREFs as its value. For example:

```
<!DOCTYPE PAPER [
<!ELEMENT SECTION (TITLE, PARAGRAPH*)>
<!ATTLIST  SECTION SEC-ID  ID  #IMPLIED>
<!ELEMENT  CROSS-REFERENCE  EMPTY>
<!ATTLIST  CROSS-REFERENCE  TARGET  IDREF  #REQUIRED>
... ]>
<PAPER>
<SECTION  SEC-ID="java.features"> <TITLE> Java's Best
features </TITLE>  ...  </SECTION>
...
To refresh your memory, see the section titled <CROSS-REFERENCE
TARGET="java.features" /> </PAPER>
```

- An ENTITY attribute is used to refer to an unparsed external entity.

```
<!DOCTYPE BOOKREVIEW [
...
<!ATTLIST BOOK COVER ENTITY #REQUIRED>
<!NOTATION GIF SYSTEM "apps/gifview.exe ">
<!ENTITY  java-book1  SYSTEM
   "http://www.sellbooks.com/java/book1.gif" NDATA GIF>
]>
<BOOKREVIEW>
<BOOK  cover = "java-book1"> ... </BOOK>
</BOOKREVIEW>
```

You may also declare an attribute to refer to one or more entities using the ENTITIES designation.

- A NOTATION attribute type is used to specify that an attribute value is one of several declared NOTATIONS. For example:

```
<!ATTLIST COVER_IMG
          type NOTATION (GIF|JPEG|BMP) "GIF">
```

After you declare your attributes and assign them to an appropriate type, you can use attributes in your document. The values assigned to those attributes are modified by a process called “normalization,” which is discussed next.

Attribute Value Normalization and Whitespace Handling

Normalization and whitespace handling are detailed processes for handling specific text processing situations. This type of fine granularity is the basis of a good standard.

Attribute Value Normalization

Element attributes are `name="value"` pairs; however, the value between the quotes is first passed through a process called normalization. Here are the steps in the normalization process:

- Surrounding quotes are stripped out.
- Character references are replaced with their corresponding characters. For example, `©` would be replaced with a copyright symbol.
- General entity references are replaced with their corresponding text. This is a recursive process, which means that if the replacement text also contains references, they are replaced, and so on.
- Whitespace characters (carriage return, line feed, tab and space) in attribute values are replaced by spaces. Also, the sequence CR-LF is replaced by a single space.
- If an attribute type is anything other than `CDATA`, leading and trailing spaces are removed. Also, if using tokenized types, spaces between tokens are collapsed to a single space.

It is important to remember the distinction between unnormalized attribute value text and attribute value data (after normalization). For example:

```
<GRAPHIC  ALTERNATE-TEXT="This is a picture of
                             a penguin dancing.">
The attribute value is normalized to
This is a picture of a penguin dancing.
```

Whitespace Handling

You may remember that in Chapter 2 we contrasted XML to HTML in its treatment of whitespace. Whereas HTML disregarded whitespace, XML preserved whitespace in your document content. To be technically accurate, the specification requires that an XML processor (usually a parser) pass whitespace on to the application (the consumer program of the data). The application then can determine whether whitespace is significant. The specification provides a special attribute called `xml:space` that can be attached to any element in order to specify the proper treatment of whitespace to the application. Here is the form of the `xml:space` attribute:

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

The `elemName` in the general form is any element you want to attach the element to. By convention, the attribute applies to that element and its children elements. The value `'preserve'`

specifies that the application should preserve all whitespace. The value `'default'` indicates that the application's default processing for whitespace is acceptable (whatever that may be).

Another aspect of handling whitespace across heterogeneous platforms is the processing of end-of-line characters. The problem is that there are three widespread methods for handling end-of-line: Mac OS uses a carriage return (CR), UNIX uses a line feed (LF) and Windows uses a carriage return line feed (CR-LF) sequence. The XML specification requires that the XML processor convert any of the stated conventions to a single LF to signify end-of-line.

Any and Mixed Element Content Models

As previously stated, element type declarations start with the literal string `<!ELEMENT` followed by an element name and then a content specification:

```
<!ELEMENT html (head, body) >
```

The content specification can be one of four types: `EMPTY`, `ANY`, mixed content, or element content. The element content model is the most common. The `EMPTY` content model is for empty elements.

The `ANY` content model allows an element to contain any character data or child elements. This is a completely unstructured content specification and therefore is rarely used.

A mixed content element may contain character data, optionally interspersed with child elements. Here is the grammar for the mixed content specification:

```
Mixed ::= '('  S?  '#PCDATA'  (S?  '|'  S?  Name) *  S?  ')*'
              |  '('  S?  '#PCDATA'  S? ')'
```

This grammar states that you can either have the literal `#PCDATA` followed by zero or more child element names or just have the literal `#PCDATA` by itself. `PCDATA` stands for parsed character data.

```
Example 1:  <!ELEMENT  NAME  (#PCDATA) >

Example 2:  <!ELEMENT  paragraph  (#PCDATA|quote|reference)* >
```

`CDATA` Sections

A `CDATA` section is used in a document when you do not want the content to be treated as markup. The most obvious example of using a `CDATA` section would be to pass XML markup into an application (instead of having it parsed as markup). A `CDATA` section starts with the string `<![CDATA[` and ends with the string `]]>`. For example:

```
<![CDATA[<TITLE> an XML example </TITLE>]]>
```

Another possible use of a CDATA section would be to pass source code to an application without having to use character references for reserved characters like the < or > symbol.

NOTE

A CDATA section is only allowed where #PCDATA is allowed in your XML document.

Conditional Sections

Conditional sections can only occur in the external subset of the document type declaration and in external entity references from the internal subset. A conditional section allows you to turn on and off a series of markup declarations. There are two keywords used with conditional sections: INCLUDE and IGNORE.

A conditional section may include one or more complete declarations, comments, processing instructions, or nested conditional sections. If the keyword used is INCLUDE, the section is processed. If the keyword is IGNORE, the section is not processed.

Here is an example of using conditional sections:

```
<! [INCLUDE [
<!ELEMENT  article (title, section+, references*)>
] ]>
<![IGNORE [
<!ELEMENT  article  (title, section+)>
]]>
```

This is very useful for turning on and off parts of a DTD during development. You can use an entity for the keyword of a conditional section. The processor will replace the reference before determining whether it should include or ignore the section. For example, the document could be rewritten like this:

```
<!ENTITY % editor "INCLUDE">
<!ENTITY % author "IGNORE">
<! [%editor [
<!ELEMENT  article (title, section+, references*)>
] ]>
<![%author [
<!ELEMENT  article  (title, section+)>
]]>
```

Processing Instructions

A processing instruction is used to pass additional information to one specific processing application without changing the way the document is processed by other applications. In general,

processing instructions should be used infrequently. The format of a processing instruction is the literal `<?` followed by a name (the name of the target application), followed by any text and ending with the literal string `?>`.

NOTE

The name of the application in a processing instruction may not be any variation of the letters *XML*.

Here is an example to change the font of the first word of a paragraph. You may have to do this if you are using someone else's DTD that does not have markup for something you want to do. For example:

```
<SECTION> The man stood on the beach.
<p> <?EZFormat Font="24Pt"?> Hey! <?EZFormat endFont ?>
</SECTION>
```

Another reason for processing instructions could be for sending special commands to a CGI program processing the XML prior to passing it to a client.

XML uses a special processing instruction for attaching XSL stylesheets to a document instance.

```
<?xml:stylesheet
          href="http://www.mystuff.com/memo.xsl"
          type="text/xsl" ?>
```

Last, remember that the XML declaration is a form of processing instruction.

Encoding and the Standalone Document Declarations

In the primer on XML, I discussed the XML declaration and stated that it contained some literal text `<?xml`, followed by version information, an optional encoding declaration, an optional standalone document declaration, and the literal text `?>`. I will now examine the two optional parts of the XML declaration: the encoding declaration and the standalone document declaration.

The encoding declaration specifies the character set encoding for the following document. The earlier section "Character References" contains a note that defines character set encoding and how it relates to both characters and character sets. The specification requires all XML processors to support both UTF-8 and UTF-16 encoding. Support of all other encodings is optional. In the absence of an encoding declaration or a byte order mark (this allows auto-detection of a UTF-16 encoded file), the encoding must be UTF-8. Because ASCII is a subset of UTF-8,

ordinary ASCII files do not need an encoding declaration. Here are some examples of encoding declarations:

```
<?xml version='1.0' encoding='UTF-16' ?>
<?xml version='1.0' encoding='ISO-10646-UCS-2' ?>
```

A list of Internet-supported character set names can be retrieved from

```
ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets
```

NOTE

Any external parsed entity may begin with a text declaration. A text declaration is identical to an XML declaration with the exception that the version declaration is optional and minus the standalone document declaration. For example:

```
<?xml encoding='UTF-16' ?>
```

The standalone document declaration is only used rarely and is not recommend for general use. As we stated previously, a DTD can be composed of both an external subset and an internal subset. An external subset is stored elsewhere and referenced via a URI. A standalone document declaration declares whether an application needs to fetch the external subset of the DTD to process the document correctly. For example:

```
<?xml version="1.0"  standalone="yes" ?>
<!DOCTYPE  HTML  SYSTEM  http://www.xmlstuff.com/html.dtd>
<HTML> ... </HTML>
```

This example would state to a processor that the client does not need to fetch the DTD to properly process the document. It is important to note that a document is not valid unless both the external subset and internal subset of a DTD has been processed.

Another scenario for using the standalone document declaration is if multiple programs process a document but only the first one validates the document. All ensuing programs could safely skip that step.

The XML Grammar

The XML grammar is specified using an Extended Backus-Naur form (EBNF) notation. Using an EBNF defines a context-free grammar—a grammar that is independent of the context in which it is used. The notation for definitions in the grammar is

```
symbol ::=  expression
```

where `expression` defines the rule for creating the symbol on the left-hand side. This formal grammar ensures that there is no ambiguity in the XML syntax. All the legal expressions are precisely defined via EBNF.

NOTE

It is important to keep in mind that this section refers to EBNF syntax and not XML syntax. The purpose for reviewing EBNF is to give you the ability to consult the XML specification when necessary.

EBNF statements are also called production rules, because they express the way in which valid symbols are constructed or produced using other symbols or specific fixed strings.

Table 4.4 shows EBNF notations in the grammar and their meaning.

TABLE 4.4 EBNF Notations

Notation	*Description*
`(expression)`	A group expression treated as a single unit.
`#x`*N*	Where *N* is a hexadecimal integer. This notation matches a specific UCS character.
`"string"`	Matches a literal string.
`'string'`	Matches a literal string.
`A?`	Matches A or nothing; Means A is optional.
`A+`	Matches one or more occurrences of A.
`A*`	Matches zero or more occurrences of A.
`A B`	Matches A followed by B.
`A | B`	Matches A or B, but not both.
`A - B`	Matches any string that matches A but does not match B.
`[a-zA-Z]` `[#xN-#xN]`	Matches any character with a value in the ranges (inclusive).
`[^a-z],` `[^#xN-#xN]`	Matches any character with a value outside the range.
`[^abc]`	Matches any character not among the given characters.

Here is a snippet of the XML grammar in the XML specification:

```
elementdecl  ::=  '<!ELEMENT'  S  Name  S  contentspec  S?  '>'
S  ::=  (#x20  |  #x9  |  #xD  |  #xA)+
Name  ::=  (Letter  |  '_'  |  ':')  (NameChar)
```

```
contentspec ::=  'EMPTY'  |  'ANY'  |  Mixed  |  children
children  ::=  (choice  |  seq)  ('?'  |  '*'  |  '+') ?
cp  ::=  (Name  |  choice  | seq)  ('?'  |  '*'  |  '+') ?
choice  ::=  '('  S?  cp  (  S?  '|'  S?  cp  )*  S?  ')'
seq  ::=  '('  S?  cp  (  S?  ','  S?  cp  )*  S?  ')'
Mixed  ::=  '('  S?  '#PCDATA'  (S?  '|'  S?  Name)*  ?  ')*'
                   |  '('  S?  '#PCDATA'  S?  ')'
```

NOTE

See Letter and NameChar rules in the XML Recommendation. They occupy several pages and were left out for brevity.

Here is a partial translation of the production rules:

An element declaration is the literal `<!ELEMENT`, followed by a space, a legal XML name, a symbol called `contentspec`, (optionally) a space, and finally the literal `>`.

A `contentspec` is either the literal `EMPTY` or `ANY`, or the translation of the symbol `Mixed` or the symbol `children`.

A `children` symbol is translated as either a choice or a sequence followed optionally by an occurrence indicator, which is the literal `?`, `*`, or `+`.

A choice symbol is translated as the literal `(`, an optional space, a symbol called `cp` (a content particle), zero or more literals `|` with more content particles (and optional space), and a literal`)`.

Namespaces in XML

As markup languages proliferate, markup language designers will want to reuse portions of languages instead of reinventing the wheel. This poses the problem of naming collisions. For example, what if we mixed HTML tags with our own Book Review Markup Language that also had a `<TITLE>` tag?

For example:

```
<HTML>
<HEAD>
      <TITLE> Book Review Page </TITLE>
</HEAD>
<BODY>
      <BOOK>
            <TITLE>  Developing XML in Java </TITLE>
```

```
            <AUTHOR>  Michael C. Daconta   </AUTHOR>
     </BOOK>
</BODY>
</HTML>
```

If a program were to parse this document, how would the programmer know which TITLE was the book title? In order to accomplish this, element and attribute names must be universal. To create a universal name, an XML name is separated into two parts: a namespace prefix and a local part. The World Wide Web Consortium (W3C) formalized the rules for creating these universal names in the Namespaces Specification, which became a W3C Recommendation on January 14, 1999. So, rewriting the previous example using namespaces produces

```
<ht:HTML xmlns:ht="http://www.w3.org/1999/xhtml"
                 xmlns:bk="http://www.gosynergy.com/brml">
<ht:HEAD>
      <ht:TITLE> Book Review Page </ht:TITLE>
</ht:HEAD>
<ht:BODY>
      <bk:BOOK>
            <bk:TITLE>  Developing XML in Java </bk:TITLE>
            <bk:AUTHOR>  Michael C. Daconta   </bk:AUTHOR>
     </bk:BOOK>
</ht:BODY>
</ht:HTML>
```

Declaring Namespaces

A namespace is declared using an attribute whose prefix is xmlns as follows:

```
<TEST  xmlns:syn="http://www.gosynergy.com/example">
```

The value of the xmlns attribute is any Uniform Resource Identifier, which functions as the namespace name. The URI does not have to actually exist. Attributes, not just elements, can also have namespaces. As an example, we could use an HTML alignment attribute to align our book title like this:

```
<ht:HTML xmlns:ht="http://www.w3c.org/HTML/1999/html4"
                 xmlns:bk="http://www.gosynergy.com/brml">
<ht:HEAD>
      <ht:TITLE> Book Review Page </ht:TITLE>
</ht:HEAD>
<ht:BODY>
      <bk:BOOK>
            <bk:TITLE ht:ALIGN="left">  Developing XML in Java </bk:TITLE>
            <bk:AUTHOR>  Michael C. Daconta   </bk:AUTHOR>
     </bk:BOOK>
</ht:BODY>
</ht:HTML>
```

Before a prefix can be used in a document, it must be declared in the current tag or in an ancestor tag that contains the current tag. Lastly, a namespace has scope. This means that the namespace applies to the element in which it is declared and all elements within the content of that element.

How Namespaces Affect the DTD

Attribute and element names are also given as qualified names (what we called universal names) in the DTD declarations. Here is an example of the DTD for our BOOK example:

```
<!ELEMENT bk:BOOK (bk:TITLE, bk:AUTHOR?)>
<!ATTLIST bk:BOOK
          xmlns:bk CDATA #FIXED
                   "http://www.gosynergy.com/brml">
<!ATTLIST bk:BOOK
          bk:pages CDATA #IMPLIED>
```

You should understand that to keep backward compatibility with SGML (which allows a colon as part of an SGML name), this is really just a syntactic trick to separate one name into two parts. Therefore, to validate the document, all names in the DTD must be modified to include the prefix part in each element and attribute declaration. Here is another example of a simple markup language that uses namespaces:

```
<?xml version="1.0" ?>
<!DOCTYPE slf:entries [
<!ELEMENT slf:entries (entry)* >
<!ELEMENT slf:entry (field)* >
<!ELEMENT slf:field (#PCDATA) >
<!ATTLIST slf:entries
          xmlns:slf CDATA #FIXED "http://www.gosynergy.com/slf">
<!ATTLIST slf:entry
          slf:type (general|security|fatalerror|
          error|warning|info|trace)   #REQUIRED
          slf:source CDATA #IMPLIED>
<!ATTLIST slf:field
          slf:name CDATA #REQUIRED >
]>

<slf:entries>

<slf:entry slf:type = 'trace'>
<slf:field slf:name='timestamp'>January 24, 2000 12:21:13 PM EST</slf:field>
<slf:field slf:name='class'> java.lang.Exception</slf:field>
<slf:field slf:name='method'>&lt;init&gt;</slf:field>
<slf:field slf:name='message'>View</slf:field>
</slf:entry>
```

```
<slf:entry slf:type = 'trace'>
<slf:field slf:name='timestamp'>January 24, 2000 3:51:48 PM EST</slf:field>
<slf:field slf:name='class'>GOV.dia.mditds.audit.AuditManagerApplet</slf:field>
<slf:field slf:name='method'>actionPerformed</slf:field>
<slf:field slf:name='message'>View</slf:field>
</slf:entry>

</slf:entries>
```

Applying Namespaces

In order to remove the burden of redundant typing, the specification allows a default namespace. A default namespace will apply to the current element where the namespace is declared if it does not have a prefix and to all subelements that do not have a prefix. So the example could be rewritten:

```
<HTML xmlns="http://www.w3.org/1999/xhtml"
        xmlns:bk="http://www.gosynergy.com/brml">
<HEAD>
      <TITLE> Book Review Page </TITLE>
</HEAD>
<BODY>
      <bk:BOOK>
            <bk:TITLE ALIGN="left">  Developing XML in Java </bk:TITLE>
            <bk:AUTHOR>  Michael C. Daconta   </bk:AUTHOR>
     </bk:BOOK>
</BODY>
</HTML>
```

In this example, the HTML namespace is the default namespace for all tags that do not have a prefix. A namespace can be overridden by another namespace declaration with the same namespace attribute name (either `xmlns` or `xmlns:name`). Lastly, the default namespace can be set to the empty string. This has the same effect, within the scope of the declaration, of there being no default namespace.

Parser Support for Namespaces

At the time of this writing, SAX is being extended to incorporate namespace support with a new set of interfaces referred to as *SAX2*. SAX2 will support namespace processing by default. This means that every element and attribute will be reported with a two-part name. The new API for an element is as follows:

```
public void startElement (String uri, String localName,
                               String rawName, Attributes atts)
       throws SAXException;
```

```
public void endElement (String uri, String localName, String rawName)
       throws SAXException;
```

NOTE

At the time of this writing, SAX2 is not widely supported. Go to the URL `http://www.megginson.com/SAX/` for more information.

The XLink Specification

The popularity of the World Wide Web is directly related to the mass appeal of hypertext links. In fact, a new phrase was coined to describe the process of moving between hyperlinked documents—*Web surfing*. Despite their popularity, current Web links are considered primitive in comparison to other linking specifications like those in the Hypermedia/Time-based Structuring language (HyTime) and the Text Encoding Initiative (TEI) guidelines. The XLink specification takes into account the advances of these predecessors to provide the capability to link XML documents.

NOTE

At the time of this writing, the XLink specification is still a candidate recommendation. The latest specification can be found at `http://www.w3c.org/TR`.

Comparison to HTML Hyperlinks

Hyperlinking has two basic components: linking and addressing. Linking is declaring a relationship between two objects. Addressing is a method for finding an object you want to associate with another object. In HTML, the `A` tag stands for *anchor*, the term in HTML for a resource. The `A` element describes a link and its `HREF` attribute points to the destination resource. The source of the link is the text in the content of the `A` element. For example:

```
<A HREF="http://java.sun.com"> Java Technology Home </A>
```

An HTML link is analogous to a simple link in XLink. Here is an implementation of a simple link using XLink:

```
<MYLINK  xlink:type="simple"
         xlink:href="http://java.sun.com"> Java Home </MYLINK>
```

NOTE

All XLink attributes and elements can only be used after you have declared the `xlink` namespace. For example:

```
<MYDOC xmlns:xlink="http://www.w3.org/1999/xlink/namespace/ ">
```

Link Types

A link type can be one of seven values: `"simple"`, `"extended"`, `"locator"`, `"arc"`, `"resource"`, `"title"`, or `"none"`. Before we explain each type, it is important to note that the value of the type attribute may be inferred by the application. In other words, the `xlink:type` attribute may be a `#IMPLIED` attribute whereby the processing application can infer its meaning from the other attributes available. This provides the flexibility for an application to treat an element as a link only under certain circumstances based on the value of other non–link-related attributes.

The definitions for each link type are

- `simple`—A constrained link between two resources that is functionally identical to an HTML `A` element.
- `extended`—A more powerful type of link that allows one-to-many connections and bidirectional traversal.
- `locator`—Identifies an element that refers to a remote resource that is participating in a link.
- `arc`—For use with an extended link in order to supply traversal, behavior, and semantic attributes for one traversal of the link (for one to-from combination).
- `resource`—For use with an extended link to specify local resources that are participating in the link.
- `title`—Both extended and locator type elements can have multiple human-readable titles by using any number of title type elements. One potential use of this is for internationalization.
- `none`—Denotes the element as a non-XLink element. This allows an element to be conditionally treated as an XLink.

Link Attributes

There are four categories of XLink attributes: locators, arc ends, behavior, and semantics. Locator attributes define where a remote resource is located. Arc ends define the context of a link traversal (like direction). Behavior attributes define how the link is activated and what

action should be taken with the resource it refers to. Semantic attributes give additional information about the link. Each category will have one or more attributes. All the attributes discussed later must be prepended with the `xlink` prefix.

There is only a single `Locator` attribute, which is `href`. The value of the `href` attribute must be a valid URI.

There are two arc end attributes: `from` and `to`. The values for both attributes must be an ID in an XML document. The intent of the `from` and `to` attributes are to provide contextual information to the processing application.

There are two attributes for specifying behavior: `show` and `actuate`. In contrast, HTML link (anchor) behavior is hardwired. The HTML link behavior is to activate the link based on a user click and replace the current document with the remote resource. The `show` attribute describes what action occurs when a link is traversed. The `actuate` attribute describes when a link traversal should occur. The `show` attribute may take one of four values: `embed`, `replace`, `new`, or `undefined`. Here are definitions for those four values:

- `embed`—The designated resource should be integrated in the body of the resource at the start of the link.
- `replace`—The designated resource should replace (for the purposes of display or processing) the resource at the start of the link.
- `new`—The designated resource should be displayed in a new window.
- `undefined`—The behavior of the application traversing the link is unconstrained by the XLink specification.

The `actuate` attribute may take one of three values: `onLoad`, `onRequest`, or `undefined`.

- `onLoad`—The link should be traversed automatically as soon as the starting resource is loaded.
- `onRequest`—The link should be traversed only on request (like a click) from the user.
- `undefined`—When the link is traversed is unconstrained by the XLink specification. The application is free to use other cues.

Replace and user are the behaviors we are familiar with in HTML links. For example:

```
<A xlink:type="simple" xlink:show="replace" xlink:actuate="onRequest"

   xlink:href="http://www.mysite.com"> This is my Site! </A>
```

Here is another example that would place the target resource into a separate window:

```
<NEWLINK xlink:type="simple" xlink:show="new" xlink:actuate="onRequest"
xlink:href="http://www.mysite.com"> This is my Site! </NEWLINK>
```

There are attributes associated with semantics: `role` and `title`. The role is a link attribute that allows a free-form description of the purpose of the link. The `title` attribute provides human-readable text describing the link. The title is useful for presentation of the link.

The inline attribute can only have the value "true" or "false." If a link is inline, its contents count as the local resource of the link. An out-of-line link is one that is completely outside the resource it is linking.

There are constraints on where attributes can occur. Table 4.5 shows where attributes can occur by type. The columns are `xlink:types` and the rows represent attributes. An X indicates that the attribute is allowed within an element of that `xlink:type`.

TABLE 4.5 Attribute Placement Constraints

	simple	extended	locator	arc	resource	title
type	X	X	X	X	X	X
href	X		X			
role	X	X	X	X	X	
title	X	X	X	X	X	
show	X	X		X		
actuate	X	X		X		
from				X		
to				X		

Extended Links

An extended link is more powerful than a simple link and has these features:

- Can connect any number of resources
- Can create links to and from documents from outside the documents they are linking (create them out-of-line)

In order to create multi-ended links, an extended link separates the source from the target of the link by using two subelement types: locator and arc. Here is an example of an extended link:

```
<TOPPICK  xlink:type="extended">
  <book xlink:type = "locator" xlink:href="book1.html" xlink:role="original" />
```

```
  <book xlink:type = "locator" xlink:href="book2.html" xlink:role="sequel" />
  <magazine xlink:type = "locator"  xlink:href="article.html"
            xlink:role="review" />
</TOPPICK>
```

Locator type elements can also have `title`, `show`, and `actuate` attributes. Locators are very similar to simple links. You can add attribute value defaults to the DTD to reduce the number of `xlink` attributes needed for a particular element.

When referring to resources, XML links can use XPointers, which are URIs that can refer inside an XML document. XPointers are discussed in the next section.

The XPointer Specification

XPointer is a complex specification for addressing into the internal structures of XML documents. In XLink, the resource can be referred to by a Uniform Resource Identifier (URI). A URI is a URL followed by an optional query and then an optional fragment identifier. XPointers are fragment identifiers that can be used in conjunction with a URL.

NOTE

At the time of this writing, the XPointer specification is a candidate recommendation. The latest specification can be found at `http://www.w3c.org/TR`.

XPointers operate on the tree structure defined by the elements and markup of an XML document. From the discussion of the Document Object Model in Chapter 3, you should know that an XML document contains seven types of nodes: root nodes, element nodes, text nodes, attribute nodes, namespace nodes, processing instruction nodes, and comment nodes. The purpose of an XPointer is to refer to a particular portion of this tree, sometimes in relation to another part. In general, XPointers select a portion of the tree with axes and predicates. An axis selects a node or group of nodes in an XML document. A predicate tests either the selected nodes or nodes relative to the selected nodes. The XPointer specification builds on another specification, called XPath, that is a common syntax used by both XPointer and the extensible stylesheet language transformation (XSLT) specification, which is discussed in Chapter 5, “Java and the Extensible Stylesheet Language (XSL).”

XPath

XPath defines a language for creating expressions that operate on an XML document tree. The most important type of expressions are location paths. There are two types of location paths:

absolute and relative. A location path consists of a set of location steps separated by a `/`. A location step has three parts: an axis, a node-test, and zero or more predicates. Here is an example of an absolute location path that selects the chapter child (or children) with a title attribute that has the value `Introduction`:

```
xpointer(/child::chapter[attribute::title='Introduction'])
```

In the example, the `/` is the absolute location for the root of the document. The term `child` is the axis. The double colon (`::`) separates the axis from the node-test. The node-test is the term `chapter`. The predicate is enclosed in quotes. For absolute location path, XPath provides `/` for the root and `id("name")` to locate a specific element with a unique ID.

Here is another example:

```
/descendant::para
```

This location path selects all the `para` elements in the document. In this example, `descendant` is the axis and `para` is the node-test.

An axis works in respect to a context node. A context node is defined either by an absolute location or a previous relative location step. The following keywords are the available axes:

- `child`—Identifies a child node of the context node.
- `descendant`—Nodes appearing anywhere in the content of the context node.
- `parent`—Identifies a parent node of the context node.
- `ancestor`—Element nodes containing the context node.
- `preceding`—Nodes before the location source.
- `following`—Nodes after the location source.
- `preceding-sibling`—Identifies sibling nodes sharing their parent with the location source that appears before the location source.
- `following-sibling`—Identifies sibling nodes sharing their parent with the location source that appears after the location source.
- `attribute`—Attributes of the context node.
- `namespace`—Namespaces of the context node.
- `self`—The context node.
- `namespace`—Contains the namespace nodes of the context node; the axis will be empty unless the context node is an element.
- `descendant-or-self`—Contains the context node and the descendants of the context node.
- `ancestor-or-self`—Contains the context node and the ancestors of the context node; thus, the ancestor axis will always include the root node.

An axis is either a forward axis or a backward axis. If the axis produces the context node and nodes after it then it is a forward axis. If the axis produces the context nodes and nodes before it (higher in the tree) then it is a backward axis.

A node-test filters nodes from an axis if those nodes do not meet certain criteria. Here are the possible node-tests:

- A qualified name—This will filter nodes if they exactly match the name. For example, `child::para` will return all the `para` elements that are children of the current node. The qualified name may include a namespace.
- One of three type tests: `comment()`, `text()`, and `processing-instruction()`—These tests return the node if it matches the type.
- `An asterisk (*)`—This is a wildcard that returns all nodes in the axis.
- An asterisk as the `localpart` of a fully qualified name—For example `child::bk:*` will return all the child elements that are part of the `bk` namespace.

A predicate filters a node-set with respect to an axis to refine the selection. Predicates evaluate to a Boolean value (true or false). There is a core function library that can be used in predicates. Table 4.6 presents the available functions in the core function library.

TABLE 4.6 XPath Core Function Library

Function Prototype	*Description*
`number last()`	Returns a number equal to the context size.
`number position()`	Returns a number equal to the context position.
`number count(node-set)`	Returns the number of nodes in the node-set argument.
`node-set id(object)`	Selects elements by their unique ID.
`string local-name (node-set?)`	Returns the local part of an expanded name.
`string namespace-uri (node-set?)`	Returns the namespace part of an expanded name.
`string name(node-set?)`	Returns the qualified name of the first node in the node-set.
`string string(object?)`	Converts an object to a string.
`string concat (string, string, string*)`	Returns the concatenation of its arguments.
`boolean starts-with (string, string)`	Returns true if the first argument string starts with the second argument string.

TABLE 4.6 Continued

Function Prototype	*Description*
`boolean` *`contains`* `(string, string)`	Returns true if the first argument string contains the second argument string.
`string` *`substring-before`* `( string, string)`	Returns the substring of the first argument string that precedes the first occurrence of the second argument string.
`string` *`substring-after`* `( string, string)`	Returns the substring of the first argument string that follows the first occurrence of the second argument string.
`string` *`substring`* `(string, number, number?)`	Returns the substring of the first argument starting at the position of the second argument for the length of the third argument.
`number` *`string-length`* `( string?)`	Returns the number of characters in the string.
`string` *`normalize-space`* `( string?)`	Returns the argument with leading and trailing whitespace removed and sequences of whitespace replaced by a single space.
`string` *`translate`* `(string, string, string)`	Returns the first argument string with occurrences of the second argument string replaced with the third argument string.
`boolean` *`boolean`*`(object)`	Converts its argument to a Boolean.
`boolean not(boolean)`	Returns the negation of its argument.
`boolean` *`true`*`()`	Returns true.
`boolean` *`false`*`()`	Returns false.
`boolean` *`lang`*`(string)`	Returns true if the argument matches the current value of `xml:lang`.
`number` *`number`*`(object?)`	Converts its argument to a number.
`number` *`sum`*`(node-set)`	The sum of the node-set calculated by converting the string values of the node to a number.
`number` *`floor`*`(number)`	Returns the largest integer not greater than the number.
`number` *`ceiling`*`(number)`	Returns the integer not less than the argument.
`number` *`round`*`(number)`	Returns the number that is closest to the argument and that is an integer.

Let's examine a complete example:

```
<!DOCTYPE  SCREENPLAY [
<!ELEMENT  LINES  (#PCDATA | SPEAKER  |  DIRECTOR)* >
<!ATTLIST  SCREENPLAY
           ID            ID  #IMPLIED>
<!ELEMENT  SPEAKER  (#PCDATA) >
<!ELEMENT  DIRECTOR (#PCDATA)> ]>
<SCREENPLAY  ID="Miller1">
   <SPEAKER> Linda </SPEAKER>
   You didn't crash the car, did you?
   <DIRECTOR> Willy looks irritated. </DIRECTOR>
   <SPEAKER> Willy </SPEAKER>
   I said nothing happened. Didn't you hear me?
</SCREENPLAY>
```

Now, let's create some XPointers in this document.

```
xpointer(id('Miller1')/child::SPEAKER[position() = 2])
     selects the 2nd "SPEAKER" element whose content is "Willy"

xpointer(id('Miller1')/child::text()[position() = 2])
     selects the second child  text element which
             is "I said nothing happened".
```

XML Schemas

There are two parts to the XML Schema specification: Structures and Data Types. The Structures specification describes a replacement syntax for describing XML documents to a finer granularity than is possible with a document type definition (DTD—the current method standardized with the XML 1.0 recommendation). The Data Types specification defines primitive data types that can be used in XML schema and other XML specifications like XSL and RDF.

NOTE

At the time of this writing, the XML Schema specification is still a working draft. The latest specification can be found at `http://www.w3c.org/TR`.

The purpose of an XML Schema is to define and describe a class of XML documents by using XML-compliant markup to constrain and document the meaning, usage, and relationships of the document's datatypes; elements and their content; attributes and their values; entities and their contents; and notations.

The XML Schema:Structures formalism will allow a useful level of constraint checking to be described and validated for a wide spectrum of XML applications.

XML Schema:Structures has a dependency on the data typing mechanisms defined in its companion document, XML Schemas:Datatypes, published simultaneously.

These are key definitions in the specification:

- *Instance*—An XML document whose structure conforms to some schema. Documents are associated with the schema to which they conform.
- *Schema*—A set of rules for constraining the structure and articulating the information set of XML documents.

Schema Structures

The key idea behind XML Schema is to define the vocabulary and content model of a markup language using the rules of XML. The basic features of XML Schema are listed in Table 4.7.

TABLE 4.7 XML Schema Features

Feature	*Definition*
Schema	All definitions and declarations are contained within a Schema element. Uses `<schema ...> </schema>`.
Simple Type Definition	The mechanisms for typing character data for either attribute values or element contents. Rules for this are specified in XML Schemas:Datatypes specification.
Complex Type Definition	A complete set of constraints for elements in a document. Uses `<type> </type>`.
Element Type Declaration	Associates an element name with a type. Uses `<element ...> </element>`.
Attribute Declaration	Associates an attribute name and a data type. Uses `<attribute ...> </attribute>`.
Content Type	Either a simple type or a content model.
Element Content Model	A type that constrains the contents of an element. Has specifications for sequences and grouping.
Attribute Group Definition	Ability to group a set of attributes under a name for reusability.
Deriving Type Definitions	A type may be based on another type and acquire content type and attributes from the other type.

TABLE 4.7 Continued

Feature	*Definition*
References to Schema Components Across Namespaces	Integrates definitions and declarations defined elsewhere into the schema as if they were defined/declared locally.
Unique Key and Key Reference Constraints	Provides powerful uniqueness and intradocument reference mechanisms.

Schema Datatypes

XML 1.0 does not provide any facility for rigorous type checking of data elements in an XML-compliant document. This specification defines standard data types for constraining values in element content and attributes' values.

The current specification concerns itself with scalar datatypes. A scalar is a single constrained value (formally, a value described in its entirety by magnitude).

Future versions of this specification will also cover aggregate data types like sets and bags (collections).

In this specification, a datatype has a set of distinct values, called its value space, and is characterized by facets or properties of those values and by operations on or resulting in those values. Further, each datatype is characterized by a space consisting of valid lexical representations for each value in the value space. A value space is an abstract collection of permitted values for the datatype. The lexical space for a datatype consists of a set of valid literals. Each value in the datatype's value space maps to one or more valid literals in its lexical space.

Datatypes can be broken down into several dichotomies. The first of these is atomic versus aggregate:

- *Atomic* datatypes are those having values that are intrinsically indivisible.
- *Aggregate* datatypes are those having values that can be decomposed into two or more component values.

Next is primitive versus generated:

- *Primitive* datatypes are those that are not defined in terms of other datatypes.
- *Generated* datatypes are those that are defined in terms of other datatypes.

Finally, built-in versus user-generated:

- *Built-in* datatypes are those that are entirely defined in the XML Schemas:Datatypes specification and can be either primitive or generated.
- *User-generated* datatypes are those generated datatypes whose base types are built-in datatypes or user-generated datatypes and are defined by individual schema designers by giving values to constraining facets.

Table 4.8 shows a description of primitive and generated datatypes.

TABLE 4.8 Primitive and Generated Datatypes

Datatype	*Description*
`string`	UCS characters of some specified length.
`boolean`	A binary-state value.
`binary`	Sequence of bytes.
`uriReference`	A uniform resource locator.
`language`	Represents natural language identifiers as defined by RFC 1766
`ID`	From XML 1.0 spec.
`IDREF`	From XML 1.0 spec.
`IDREFS`	From XML 1.0 spec.
`ENTITY`	From XML 1.0 spec.
`ENTITIES`	From XML 1.0 spec.
`NMTOKEN`	From XML 1.0 spec.
`NMTOKENS`	From XML 1.0 spec.
`NOTATION`	From XML 1.0 spec.
`name`	An XML name as defined by the XML 1.0 spec.
`QName`	A qualified XML name as defined by the XML Namespace recommendation.
`NCName`	NCName represents XML "non-colonized" names as defined by the XML Namespace recommendation.
`integer`	Whole numbers.
`PositiveInteger`	Derived from `nonNegativeInteger` by fixing the value of `minInclusive` to be 1.
`nonPositiveInteger`	Negative integers where the value of `maxInclusive` is fixed at 0.
`negativeInteger`	Negative integers where the value of `maxInclusive` is –1.
`nonNegativeInteger`	Derived from integer by fixing the value of `minInclusive` to be 0.

TABLE 4.8 Continued

Datatype	*Description*
long	long is derived from integer by fixing the values of maxInclusive to be 9223372036854775807 and minInclusive to be –9223372036854775808.
int	int is derived from long by fixing the values of maxInclusive to be 2147483647 and minInclusive to be –2147483648.
short	short is derived from int by fixing the values of maxInclusive to be 32767 and minInclusive to be –32768.
byte	byte is derived from short by fixing the values of maxInclusive to be 127 and minInclusive to be –128.
unsignedLong	Derived from nonNegativeInteger by fixing the values of maxInclusive to be 18446744073709551615.
unsignedInt	Derived from unsignedLong by fixing the values of maxInclusive to be 4294967295.
unsignedShort	Derived from unsignedInt by fixing the value maxInclusive to be 65535.
unsignedByte	Derived from unsignedShort by fixing the value maxInclusive to be 255.
decimal	Numbers with an exact fractional part.
real	Floating-point numbers expressed with a mantissa and an exponent.
float	IEEE single-precision 32-bit floating point type.
double	IEEE double-precision 64-bit floating point type.
date	Date as a string as defined in ISO 8601.
month	A timePeriod that starts at midnight on the first day of the month and lasts until the midnight that ends the last day of the month.
year	A timePeriod that starts at the midnight that starts the first day of the year and ends at the midnight that ends the last day of the year.
century	A timePeriod that starts at the midnight that starts the first day of the century and ends at the midnight that ends that last day of the century.
time	Time as a string as defined in ISO 8601.
timeInstant	Represents a specific instant of time.
timePeriod	A period of time as a string as defined in ISO 8601.

TABLE 4.8 Continued

Datatype	*Description*
`timeDuration`	Represents a duration of time as defined in ISO 8601.
`recurringDay`	A specific day that recurs within a specific `timeDuration`.
`recurringDate`	A specific date that recurs.

Strings can be constrained using either picture elements (from COBOL) or regular expressions.

A Sample Schema

To demonstrate and compare schemas in relation to DTDs, I present a schema for our Address Book Markup Language (ABML) that we created a DTD for in Chapter 1:

```
<schema targetNameSpace="http://www.gosynergy.com/abml"
        xmlns = "http://www.w3.org/TR/1999/WD-xmlschema-1-19991217"
    xmlns:abml = "http://www.gosynergy.com/abml" >

<element name="ADDRESS_BOOK" type = "ADDRESS_BOOK_TYPE" />

<type name="ADDRESS_BOOK_TYPE">
    <element name="ADDRESS" type="ADDRESS_BOOK_TYPE" minOccurs="1"
             maxOccurs="*" />
</type>

<type name="ADDRESS_TYPE" >
    <element name="NAME" type="string" />
    <element name="STREET" type="string" />
    <element name="CITY" type="string" />
    <element name="STATE" type="string" />
    <element name="ZIP" type="string" />
</type>
</schema>
```

Summary

This chapter covered a lot of ground. Some of the specifications discussed are completed and others are still works in progress. The five primary categories discussed were advanced markup, namespaces, XLink, XPointer, and XML Schemas.

The advanced markup section covered all the areas of the XML 1.0 specification left out in Chapter 1. Here is a brief description of the topics covered in this section:

- Character references enable you to represent any Unicode character in your XML document.

- Entities allow you to abbreviate some replacement data and refer to that data via an entity reference. There are several categories of references to include general, parameter, internal, external, parsed, and unparsed.
- All attributes are typed in order to constrain the values that may be assigned to them. The legal attribute types (in a DTD) are `CDATA`, `Enumeration`, `NOTATION`, `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES`, `NMTOKEN`, and `NMTOKENS`.
- Attribute values are normalized before being passed on to the processing application.
- The `xml:space` attribute can be attached to any element to determine how whitespace should be handled.
- Element content specifications describe how subelements may be nested. We discussed the ANY and mixed content models.
- `CDATA` sections allow you to pass raw text (even XML) on to your processing application without it being treated as XML.
- A processing instruction may be used to pass additional information on to one specific application.
- The standalone document declaration is used to signify whether the XML processor is required to fetch the DTD.

Namespaces are a W3C recommendation to create element and attribute names that are globally unique. A unique name is created by separating an XML name into two parts: a prefix and a local part. The prefix is further mapped to a URI.

The XLink specification defines how to create links in XML documents. Links are created via attributes in the XLink namespace. XLink defines two types of links: simple and extended.

The XPointer and XPath specifications define a syntax to refer to a specific element or set of elements inside of an XML document. XPointers use location path expressions. A location path is a series of location steps separated by a `/`.

XML Schemas define a new syntax for describing the structure and datatypes of a class of XML documents. Unlike DTDs, XML Schemas are a markup language that conforms to XML syntax. XML Schemas also have a larger set of built-in datatypes than DTDs to include float, int, date, and uriReference.

Suggested for Further Study

1. Create a DTD for a Bookmark List Markup Language (BLML). A bookmark list can contain folder elements and bookmarks. A folder element should have a name attribute. A bookmark element must have the following information associated with it: name, URI, and comment. The DTD must use the following entities:

a. A general entity (that is, an abbreviation for your full name as the author of the comments).

b. A parameter entity (that is, the name attribute is common between the folder and the bookmark elements).

2. Here's a suggestion for an advanced DTD study. Define an element called `STUFF` with the following constraints:
    - The element can have A, B, and C subelements.
    - The element must have at least one of those subelements.
    - The element cannot have any duplicates. For example, if you have an A, you cannot have a second A.

    (HINT: a good solution will use only three particles.)
3. Add a namespace to your BLML DTD and in an instance of a BLML document.
4. Create a schema for the Bookmark List Markup Language.

Further Reading

XML Specification Guide. Ian S. Graham and Liam Quin. 1999, John Wiley & Sons, Inc. This book is an authoritative reference that explains every line of the XML specification in detail.

XML Unleashed. Michael Morrison. 1999, Sams Publishing. This book covers XML technology broadly, including DTDs, XSL, and XPointers, and manipulating XML with Java and JavaScript. The book includes XML applications involving e-commerce, database access, Web management, real estate, and healthcare as well as reference material on SMIL, the XML-based language for Web multimedia.