Chapter 1

Creating XML Documents

The eXtensible Markup Language (XML) is a text- and data-formatting language that, like HTML, has a tag-based syntax. At first glance, in fact, it looks a lot like HTML, but it's capable of doing much more: Not only can you prescribe text styles, but the power of XML comes from its capability to define data types for cross-platform communication. Furthermore, XML's *extensibility* enables you to create your own tags for developing applications for specific needs.

The basis of XML is the XML document. In its most essential form, it is a text file with the `.xml` extension that contains text, data, and the XML tags. XML does no data processing, however; for that, you need to employ an XML processor, or *parser*. Parsers are compiled applications developed in any number of programming languages, such as C or Java. Parsers come in many flavors, and sometimes are embedded in other applications, such as Internet Explorer 5 (IE 5) and above. But first, you need to learn how to create a basic XML document.

The simplest XML documents can be created using a text editor and just a few short lines of code. The most complex

Adapted from ***Mastering™ XML, Premium Edition***, by Chuck White, Liam Quin, and Linda Burman

ISBN 0-7821-2847-5 1,155 pages $49.99

XML documents, however, could not possibly be developed without the help of powerful software. In fact, many XML documents are extracted from a database or through other processes.

This doesn't, however, preclude your understanding the intricacies of XML. The contrary is probably true, because the more that is hidden from you, the harder it is to understand what is happening under the hood.

This chapter explores how to build an XML document. First, we'll begin with a simple XML document. Next, we'll take you through the basic syntax of XML. Then, we'll break an XML document into its components. We'll look briefly at namespaces, and then we'll help you understand how to choose which rule-based system to use—document type definitions (DTDs) or schemas. Finally, we'll build another simple XML document, with enough added complexity to get you ready for the rest of the book.

Creating an XML Document

Writing your first XML document is so easy, we're going to dive right in and write one. We're even going to write a simple style sheet to get you to see it in a Web browser.

NOTE

You'll need IE 5 or above or Netscape 6 or above to view this example.

This basic XML document contains one element, called `Basic`. This is really the essence of XML: the ability to define your own meaning and structure to a document. So open your favorite text editor and type the contents of Listing 1.1, beginning with `<?xml version="1.0"?>`.

Listing 1.1: A Basic XML Document

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Basic.xsl"?>
<Basic>Hello World</Basic>
```

Next, save the file as `HelloWorld.xml`. If you are using Notepad in Windows, in the Save As dialog box be sure to select All Files from the Save As Type drop-down list. If you don't, Notepad will save the file with

a `.txt` extension, and you may not be able to open the document in an XML processor. You can also wrap the filename in quotes as an alternative. If you are on a Macintosh, be sure to save the file with an `.xml` extension. Most Macintosh programs default to saving files without an extension of any kind, so you will need to account for this.

Next, let's develop a simple style sheet so we can view the file in a browser. This isn't necessary, because XML parsers, including the one that comes with IE 5, will parse the document and reveal its structure. But we want to give you something here that looks familiar, and most people have by now seen a simple Web page.

The next step then is to create our simple style sheet. Type the contents of Listing 1.2, beginning with `<?xml version="1.0" ?>`.

Listing 1.2: A Basic XSL(T) Document

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL
➥/Transform">
<!– xmlns:xsl="http://www.w3.org/TR/WD-xsl" for
most versions of Internet Explorer 5 –>
        <xsl:template match="/">
           <html>
              <head>
                  <title>A Basic Stylesheet
                  </title>
               </head>
               <body>
                  <xsl:value-of select="/"/>
               </body>
           </html>
        </xsl:template>
</xsl:stylesheet>
```

Next, save this file as `Basic.xsl` in the same directory or folder as `HelloWorld.xml`. The browser will need to be able to access it in order to view `HelloWorld.xml`.

If you have Netscape 6, you can choose Open File from the File menu and navigate to the directory in which you saved `HelloWorld.xml`. Open the file in Netscape, and the screen shown in Figure 1.1 should appear on your desktop.

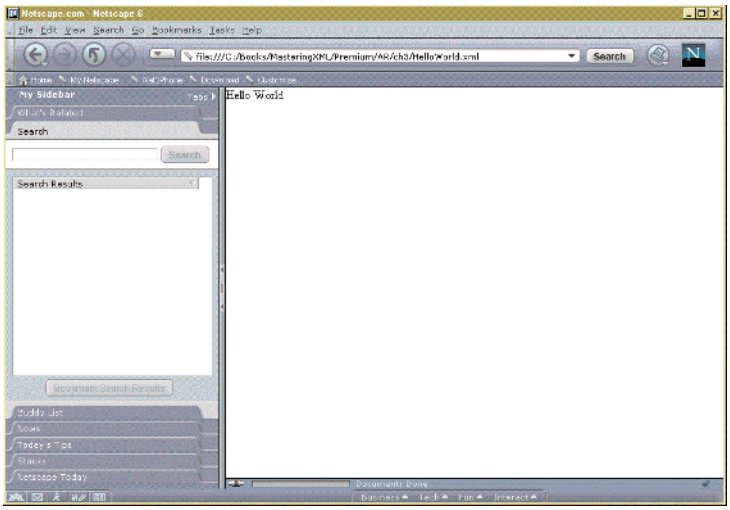

FIGURE 1.1: The file Basic.xml as rendered by Netscape 6

NOTE

You don't need to build a style sheet in Netscape. Netscape will render the document using built-in style sheets based on Cascading Style Sheets (CSS), which is a style sheet language used by some Web browsers.

If you have IE 5, choose File ➢ Open to navigate to `HelloWorld.xml`. You may need to swap the portion of the XSL code that reads `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"` for the commented code that reads `xmlns:xsl="http://www.w3.org/TR/WD-xsl"` and resave `Basic.xsl` if IE 5 doesn't seem to render anything. Just replace one with the other doing a copy and paste. There's no need to change the code between the <!–– and ––> comment tags. (See the "Miscellaneous Statements" section later in this chapter for more information about using comments.) If you found that step necessary, refresh your browser after resaving `Basic.xsl`. The page should now display, looking like the screen in Figure 1.2.

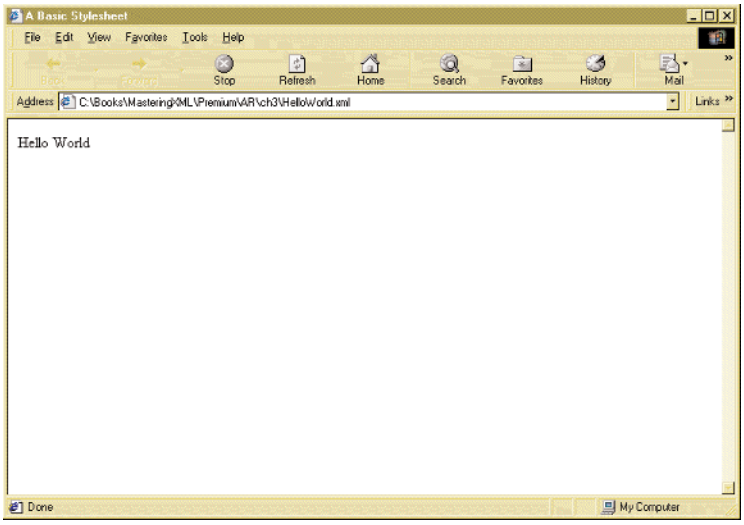

FIGURE 1.2: The file HelloWorld.xml as rendered by Internet Explorer 5

Two Kinds of "Legal" XML

XML parsers read two types of XML documents that can, if not marked up properly, generate errors: *well-formed* and *valid*. A well-formed document is syntactically correct but isn't necessarily valid against a DTD. A valid document has been validated against a DTD.

Well-Formed XML Documents

A well-formed XML document satisfies the production rules described in the XML specification, which is defined at `www.w3.org/TR/REC-xml` and in Appendix A of this book. These rules are discussed in this chapter. An XML parser will always generate errors if it encounters an XML document that is not well formed and will never make any adjustments to try to compensate for the offending document. This is very different from an HTML browser, which will generally allow for syntax mistakes in HTML code and render a document as best it can.

A well-formed document consists of character data and markup. The markup separates the content (character data) of a document from its start tags, end tags, empty element tags, entity references, character references, comments, CDATA section delimiters, document type declarations,

and processing instructions. You'll learn what each of these are as this chapter progresses.

Valid XML Documents

A valid document is validated against a *document type definition (DTD)*, which is a set of rules that you as a developer can create to describe the semantics of an XML document's markup. Semantics tell a computer what meaning to give to the markup. Browsers know the purpose behind an HTML element such as `<H1>` because the browser is somehow made aware of the meaning. (HTML itself is described using an SGML DTD, and XHTML, an XML-conformant version of HTML, is described using an XML DTD.) An XML parser that isn't a validating parser will not check the document against the rules in a DTD and will not generate any errors if the document doesn't conform to the DTD, as long as the document is at least well formed. A validating parser will examine the XML document to see if it matches the rules as declared by the DTD with which it is associated. If the document doesn't match the rules perfectly, it isn't a conforming document, and the parser will generate an error, even if the document is otherwise a well-formed document.

Understanding General Syntax

The general syntax of an XML document is dictated by a set of rules defined by a document created by the W3C (World Wide Web Consortium). This document is called the XML 1.0 Specification, and it consists of a grammar-based set of production rules that are based on a formal notation called the Extended Backus-Naur Form (EBNF), which is used to formalize the syntax of a computer language, in addition to explicit rules that extend beyond the EBNF production rules and which must also be adhered to. EBNF is designed to be read by a machine, so at first glance it can be quite a mysterious-looking beast, but it's actually no worse than getting through a DTD. It just looks different from what you might be used to seeing.

We've eliminated EBNF entirely from the mix in this chapter and spelled out the grammatical rules of XML entirely in plain English (or, if you're reading a translated version of this book, in your own language). If you want to see the actual EBNF-based specifications, take a look at Appendix A. We also provide a guide to EBNF in Appendix A to make interpreting its syntax easier.

The Basic Rules

You need to understand a few basic rules of syntax before diving into the rest of the structure of XML documents. These rules are absolute. When we say, for example, that you need to put quotes around attribute values, we don't mean sometimes, or with the possible exception of another event, but *always*. This is one of XML's strengths. Getting used to this depends on your background. If you come from a SQL background, you'll find XML rather intuitive. If your background is HTML, the constraints may initially frustrate you. Eventually, though, you'll find that just as a database constraint is your best friend, so are the constraints in XML, and you'll learn to love them.

Case Sensitivity in XML

XML is case sensitive. This is the first rule we'll cover, and it bears repeating: XML is case sensitive. If a rule in a DTD defines an element called `foo`, you can't call it `FOO` when you use that element in your document. In addition, the element's start tag must exactly match the end tag: `<foo></foo>`, not `<foo></FOO>`. Any variation from that rule will generate an error.

A real-world example: XHTML, which is the newest version of HTML and which conforms to an XML DTD, requires, by definition, that all its elements be lowercase. There was not a small amount of wrangling over this, but the W3C group that authored the specification needed to make a decision one way or the other. If they had gone the other direction, a large group of folks who prefer lowercase would have put up substantial resistance. Either way, the W3C group in charge of XHTML was going to make people unhappy. They could also have defined both lower- and uppercase rules, but that would have made the DTD twice as big and would have made extending the spec more unwieldy.

Start and End Tags

All start tags must have end tags. End tags, without exception, must always follow start tags in XML. An end tag can appear immediately after the name of the element if the element doesn't contain content, but it must appear. The following elements have both start and end tags:

```
<Basic>Hello World</Basic>
<Basic/>
<Basic></Basic>
```

As you can see, you can create an empty element tag in two ways. By convention, most developers use the first empty element example (<Basic/>). Many developers also prefer to add a space between the tag name and the / character, which keeps it consistent with XHTML.

Start Tag Consistency

The start tag that begins a document instance must end it. This is the same thing as saying that all XML documents must contain a root element (also referred to as the document element in the Document Object Model, or DOM, which is a set of interfaces that allow you to manage an XML document programmatically). The root element is always the first element in a document instance, and a root element must contain all other elements, if any exist. The following, then, will generate an error:

```
<Basic/>
<Next></Next>
```

The following is correct:

```
<Basic>
<Next></Next>
</Basic>
```

NOTE

The root element is not the same as the *root* of the document, sometimes referred to as a *root node* in applications such as XSLT. The root of the document begins with the first character an XML processor encounters when parsing an XML document and ends at the last character.

Proper Nesting of Elements

Elements must be properly nested. Elements can't stand alone in a document (unless there is only one element, the root element). All of them need to be contained within a hierarchy of elements that begins with the root element.

The easiest way to understand this concept is to simply get into one valuable habit when marking up an XML document: When you create a start tag for an element, immediately create the end tag. Another element can exist within an element, but each element's start tag must have a corresponding end tag before another element's start tag begins.

Think of plastic storage containers in your kitchen. If you have a lot of them, you probably store some of them together, one inside another. The largest one contains the next largest one, and that one contains the next largest size, and so on. XML doesn't care how big your element is, but like with the storage containers in your kitchen, you can't place part of one element inside one element and another part inside another element. If you have three elements total in your XML document, the root element must contain the other two:

```
<Basic>
   <Next type="the next one">
      <onemore>some content</onemore>
   </Next>
</Basic>
```

This markup would be wrong if the elements weren't nested properly. The following will generate an error:

```
<Basic>
   <onemore>

      <Next type="the next one">some content
   </onemore>
      </Next>
</Basic>
```

The preceding code contains an example of an element with another element as content and an element with character data as content. You could also have an empty element mixed in:

```
<Basic>
   <onemore>

      <Next type="the next one">some content
         <br />and some more</onemore>
      </Next>
</Basic>
```

We included the extra space in the <br> element in deference to HTML browsers, which can have trouble with an empty <br> element without the space (
) but can handle it otherwise.

Reserved Markup Characters

The open angle bracket (<) and ampersand (&) are reserved for markup. Element tags must begin with the < character, and entities and character references in a document instance must begin with the & character, which means that if you use either of these characters for any other purpose, you'll generate an error. When an XML parser encounters the < character, it assumes an element or other markup statement is about to start. If it doesn't find the characters it is expecting next, an XML name followed directly by a right angle bracket (>), or a comment or processing instruction, it generates an error. Similarly, when an XML parser encounters an & character, it assumes it has encountered an entity. There are five predefined entities in XML:

< Generates the < character in character data.

> Generates the > character in character data.

& Generates the & character in character data.

' Generates the ' character in character data.

" Generates the " character in character data.

If the characters following the & character don't consist of characters that help to build one of the preceding list of entities, the parser will assume the entity was defined in the DTD or is a character reference. If the parser doesn't find that definition or the proper character reference, it will generate an error.

Character references are similar in appearance to entity references, but, depending on encoding, they don't need to be declared and they refer to specific characters (such as accented letters) using a special numbering system called Unicode. You can find a chart of character references in Chapter 4, "Understanding and Creating Entities."

Using predefined entities in place of the <, >, &, ', and " characters is called *escaping* a character. This just means you are guaranteeing their safety so that you actually do end up with the characters you're hoping for. Notice the greater-than character (>). Always escape this character even if you're pretty sure there is no less-than character (<).

Examine the following lines of code to see if you can identify the legal and illegal uses of the < and & characters. We'll identify the correct answers by referencing the code's line numbers in the paragraphs that follow.


```
1.  <!ENTITY rights "&#169;">
2.  <fragment>&rights;2001 Chuck White</fragment>
3.  <&fragment>foo</&fragment>
4.  <fragment>foo & foo</fragment>
5.  <fragment>1 < 2 </fragment>
6.  <fragment>&replacement;  &more</fragment>
7.  <fragment>&replacement;  &more;</fragment>
8.  <fragment>
    The <> operator must be escaped
    </fragment>
9.  <fragment>
    I was hoping I could create this "null" element:
    </></fragment>
10. <fragment>
    Maybe it will work if I add a space:
    < /></fragment>
11. <fragment>
    I was hoping I could create
    this "null" element: <![CDATA[</>]]>
    </fragment>
```

Line 1 correctly uses an entity reference declaration in a DTD document. The declaration says, "Replace the entity `rights` with the copyright symbol (the `&#169;` character reference)."

Line 2 correctly uses the declared entity within the document instance. If output to text using an XSLT transformation, the result would look like this: `2001 Chuck White`.

Line 3 is incorrect. It will generate an XML parser error that says something like "A name was started with an invalid character."

Line 4 is also incorrect, because you can't use the & character as part of element content when it's not following the rules we've described here. You'll get an error message similar to "White space is not allowed at this location."

You'll receive the same message if you try to parse Line 5. That's because the parser expects the < character to be a start tag.

Line 6 would have been legal if both ENTITY declarations were made in the DTD, but the author left off the semicolon at the end of the second entity, which would generate an error.

Line 7 is correct. It's a corrected version of Line 6 (assuming there is a corresponding ENTITY declaration).

Lines 8, 9, and 10 will not work either, because they are all incorrect uses of the start tag character.

Line 11 corrects Line 9 by using a CDATA section to escape the characters into raw text so that the parser does not attempt to interpret them as markup.

XML Declaration Priority

If you are using an XML declaration, it must come first. The XML declaration simply declares that a document is an XML document and describes its version. It is optional, but if you use it (and by convention you should unless you're working with a document fragment for inclusion in another XML document), it must, unequivocally, be the first statement in an XML document:

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

The XML declaration is part of the document prolog, as you will discover later, and not part of the document instance (the main body of the document that holds the data you're working with). It has no bearing on the ordering or nesting of elements and is, in fact, not an element itself. Therefore, it is not subject to the rule that dictates that a root element must contain all other elements. This is *not* an exception to any rules; it is part of the rules. Because an XML declaration does not qualify as an element, it is not subject to the rules to which elements must adhere. It is also not a processing instruction, although it looks like one. A processing instruction hands off instructions to another application. An XML declaration doesn't do that.

Quotation Marks for Attribute Values

Attribute values must be enclosed in quotation marks. In HTML, attribute values don't have to be in quotes for a browser to render a document. Not so for XML. Leave off the quotes, and an XML parser is *required* to generate an error. Whether you use single or double quotes is up to you, but be consistent at each end of the attribute value. It's okay to nest one type of quote, such as single quotes, inside another set of a different type of

quotes, such as double quotes. The following are examples of attribute values in each kind of quotes:

```
<foo myatt="1.0">
<foo myatt='1.0'>
```

Characters, Markup, and Tags

To create an XML document, you should also have an understanding of markup and character data. Each of these can be categorized as one of the following:

- Part of a name token
- A white space character
- A member of a literal string
- Markup

These are explained in the following sections.

XML Names and Name Tokens

XML *names* are important because so many parts of an XML document are bound by the rules associated with them. An XML name describes the rules that declare how an XML name can be defined. A *name token* is any mix of name characters. XML names consist of name tokens, but there are some restrictions, as noted in the following rules. There is a distinction between name tokens and XML names. Name tokens aren't used only in XML names. They can also be used to identify the data type of an attribute and to define the syntax for enumerated values in attributes (values that are declared as the only acceptable values by a DTD). The rules are as follows:

- An XML name may begin with an underscore or a letter.
- XML names can contain letters, digits, periods, hyphens, underscores, and colons.
- XML names can contain combining characters (a letter with a mark attached to it, such as the combination of an accent mark that appears directly over a letter), and extending characters (which aren't letters but rather alphabetic symbols that some languages use and that act like letters in many ways). Extending characters are not an English language phenomenon.

- XML names cannot contain any punctuation marks other than periods, colons, or hyphens.
- XML names cannot contain white space.
- XML names cannot begin with a number (or more correctly expressed, a digit), but can contain digits (yes to `<element5>`, no `<5element>` o).
- Name tokens adhere to the same rules as XML names, except that they *can* begin with a number.
- Both XML names and name tokens are case sensitive.

Examine the following lines of code to see if you can identify the legal and illegal uses of XML names. We'll identify the correct answers by referencing the code's line numbers in the paragraphs that follow.

```
1.  <fragment></fragment>
2.  <fragment5></fragment5>
3.  <5fragment5></5fragment5>
4.  <fivefragment5></fivefragment5>
5.  <five,fragment5></five,fragment5>
6.  <five;fragment5></five;fragment5>
7.  <five_fragment5></five_fragment5>
8.  <five_fragment5:frag5></five_fragment5:frag5>
9.  <!fragment></!fragment>
10. <[fragment]></![fragment]>
11. <xmlFoo/>
```

Lines 1 and 2 are correct.

Line 3 will generate an error that states an element began with an invalid character.

Line 4 is correct.

Lines 5 and 6 contain invalid characters.

Line 7 fixes Lines 5 and 6 by using a legal underscore character.

Line 8 will generate an error that says something like " Reference to undeclared namespace prefix: `five_fragment5`." Namespaces are covered at the end of this chapter.

Line 9 will generate an error that complains that a declaration has an invalid name.

Line 10 will generate an error that complains that an element began with an invalid character.

Line 11 is incorrect, because it starts with `xml`, which cannot start XML names; however, it will not generate an error.

White Space Characters

White space has different meanings in different applications, whether it be print media, code development, or HTML. The term *white space* here designates such characters as line feeds, tabs, carriage returns, and non-breaking spaces outside XML markup, which are always preserved. For example, when the following fragment of code is output into text via an XSLT transform, the space is preserved (unless steps are taken to override the default XSLT mechanism):

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


foo</fragment>
```

The preceding fragment, when transformed into text, looks like this:

```
foo            foo


foo
```

You'll get the same kind of result if you place the white space in an attribute value. So the following is perfectly acceptable, and when output, the spaces will be preserved:

```
<fragment x="1  2         5 "/>
```

However (and you'll see the logic here), if you add a space to the `fragment` element, you'll have a mess on your hands. The parser will expect an equal sign after *ment*:

```
<frag ment x="1  2         5 "/>
```

You can probably see why just by examining the code. When a white space appears within an element, that element name ends. Thus, the following works fine:

```
<fragment    ></fragment>
```

That's because as soon as the XML parser encounters the white space in the XML element name, it is satisfied that, in this case, the name is `fragment`, and it moves on to the next order of business.

Having said all that, if you were to load the above text into an HTML browser, the browser would "normalize" the text so that the spaces disappear to the point where there is no more than one space between words, and no line breaks (unless you include markup to quash this behavior). So if you are used to HTML's treatment of white space, be aware that XML acts differently.

Literal Strings

Literal strings are quoted strings that don't contain other quoted strings. They manage the content of internal entities, attribute values, and external identifiers. There are three kinds of literal strings:

- Literal strings that define entity values. These can consist of any character except the % and & characters, unless they are starting an entity reference within the literal string (in other words, the literal string that defines your entity reference can itself contain an entity reference). So `"Today &amp; Tomorrow"` is okay as part of your literal string entity reference, but `"Today & Tomorrow"` will generate an error.
- Literal strings that define attribute values, such as the following: `<fragment x="1"/>`.
- System literals that define URIs (Uniform Resource Identifiers), such as those found in entity definitions: `<!ENTITY foofrag SYSTEM "foo.txt">`.

Examine the following lines of code to see if you can identify the legal and illegal uses of string literals. We'll identify the correct answers by referencing the code's line numbers in the paragraphs that follow.

```
1.  <fragment x="y">"<markup/>"</fragment>
2.  <fragment x="1+2, 'x/3'=1, '5"/>
3.  <!ENTITY foofrag SYSTEM "foo">
4.  <!ENTITY foofrag SYSTEM "foo.txt">
5.  <!ENTITY foofrag  "foo">
```

```
6.  <fragment >"Excellent," said Holmes, "That's why she
    cried, 'Liar!'"</fragment>
```

Line 1 is fine. It consists of a simple string literal used as content within an element.

Line 2 may look like a lot of funny business is going on, but it's really okay. All the quotes within the double quotes are single quotes. The only bad thing that can really happen is when a non-XML parser encounters such a beast and is expecting a terminated string. But the XML parser itself doesn't care about such things. It only cares that literal strings be encapsulated within a pair of double quotes.

Line 3 is a mini-brain teaser. Technically, it is okay, but because it's a system literal, the parser will expect to be able to locate the named URI.

Assuming the entity is a text file, Line 4 is a much better bet. It could also be that the author of the DTD meant the entity to merely be the string, `foo`, in which case Line 5 is the correct syntax.

Line 6 is a good example of why it's important to be able to include single quotes within double quotes. This is an acceptable use of string literals.

Markup

Markup is notation that provides information to an XML parser on how to parse, or read, an XML document; which parts of a document to skip; and which parts of a document to hand off to another application. Based on that information, a parser, as part of its parsing routine, looks to see if the document is well formed. If a DTD is associated with the document, the parser will, if it is a validating parser (in other words, if it actually possesses the ability to test for it), test the document to see if it is valid. If the document fails either a well-formedness test or a validity test, the parser will return an error.

The kind of markup you encounter depends on the type of nodes, or components, of an XML document you encounter. The markup for a processing instruction is different from the markup for an element, an entity, or a comment.

As mentioned at the beginning of the chapter, the following characters or group of characters all form markup: start tags, end tags, empty element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions.

The Components of an XML Document

XML doesn't require two documents in order to accomplish something. But starting off this way does force you into thinking the way you will need to in the long term when managing XML development.

XML documents consist of two or more main document entities: the document prolog, the document instance, and, optionally, any processing instructions you might have. The prolog is like an introductory notation that gives instructions to a processor about how to handle the main part of the document, which is the document instance. Processing instructions can appear arbitrarily within either the document prolog or the document instance—it's up to you. They are for sending instructions to another processing application beyond the realm of the XML parser.

The Document Prolog

All XML documents start with a prolog, even if there is nothing in the prolog. Generally, there is something, because a prolog contributes mightily to a processing environment's capabilities and removes default processing routines that may not be wanted. A document prolog consists of the following, in the order shown:

1. An optional XML declaration
2. Zero or more miscellaneous statements
3. An optional document type declaration
4. Zero or more miscellaneous statements

The order of these is important. If you stray from the order specified, your XML parser will generate an error message.

NOTE

XML parsers are not allowed by the specification to "fix" errors in your code. All they are allowed to do is generate a message that reports the error to you.

The XML Declaration

The XML declaration is the first thing you usually will see in an XML document (the only exception is if there is nothing in the prolog). It consists of a left angle bracket, followed by a question mark character, and, with no spaces, the following three characters: `xml`. The simplest XML declaration looks like this:

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

The declaration consists of, in the order shown, the following:

1. This specific string `<?xml`
2. A required statement defining the version of XML used by the document instance
3. An optional declaration defining the encoding
4. An optional declaration describing whether the XML document is a standalone document
5. An optional white space character
6. The string value, `?>`

Here is an example of a complete XML declaration:

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
```

Many people mistake some of the statements in an XML declaration for attributes, and some XML editing applications even describe them as such in their graphical user interfaces (GUIs), but they're not attributes. An XML declaration, like the rest of the document prolog, has its own rules of syntax, quite separate from the document rules that govern a document instance. In addition, many people mistake XML declarations for a processing instruction.

XML Version Declaration There is only one version of XML: 1.0. When creating an XML version declaration, you can use single or double quotes (as long as the opening and closing quotes are either both single or both double), and you can include a white space character on either side or both sides of the = operator. The following is a legal XML declaration:

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

Naturally, there may be versions of XML in the future. So by convention, almost everyone now includes the version information in their XML

declaration so they won't need to worry about backward-compatibility issues.

Encoding Declaration This is especially useful for dealing with non-Western languages. Generally, if you are an English speaker, your operating system uses a 7-bit ASCII encoding, which is a subset of UTF-8 (a Unicode encoding), which in turn is the default encoding scheme in XML parsers.

Standalone Declaration The standalone declaration indicates whether the document has any links that make it a complete document. You can use single or double quotes (as long as the opening and closing quotes are either both single or both double).

The following is an XML prolog with an encoding declaration and a standalone declaration. The order is important. And remember, even though they look like attribute value pairs, these declarations are not attribute value pairs and need to be in the order shown:

```
<?xml version = "1.0"
encoding="UTF-8" standalone="yes" ?>
```

Miscellaneous Statements

Miscellaneous statements can include comments, which are notations describing the purpose of one or more aspects of the document and which are completely ignored by the parser because they are designed strictly for human consumption. They can also include white space and processing instructions.

Comments are simple to create in XML. They always start with the <!–– characters and end with the ––> characters. The parser always ignores everything in between. The following is an example of a comment:

```
<!–– this is a comment. Anything can go here ––>
```

The following comment is also acceptable, even though it contains characters that would generate errors in other circumstances:

```
<!–– <1 & anything else : ;  ––>
```

Document Type Declaration

The document type declaration declares which document type definition (DTD) is associated with the document instance. If the DTD is embedded within the document as a whole, its declarative statements follow. If the

DTD is linked, the declaration contains link information that tells the XML parser where to find the DTD.

Let's take a look at both embedded and linked document type declarations. Listing 1.3 shows an embedded document type declaration, and Listing 1.4 shows a linked document type definition.

Listing 1.3: Embedded Document Type Declaration

```
<?xml version = "1.0"
encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE fragment
[ <!ELEMENT fragment (#PCDATA)>
<!ENTITY foofrag  "said Holmes, ">
]>
<fragment >"Excellent," &foofrag;
"That's why she cried, 'Liar!'"</fragment>
```

Listing 1.4: Linked Document Type Definition

```
<?xml version = "1.0"
encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE fragment SYSTEM "fragment.dtd">
<fragment >"Excellent," &foofrag;
"That's why she cried, 'Liar!'"</fragment>
```

If you don't specify an absolute URI in Listing 1.4, your parser will need some other way of locating the DTD. Generally this means it exists in the same directory or a named directory relative to the root of the XML document.

The Document Instance

The document *instance* is the main part of an XML document that follows the prolog. It contains the content of the XML document, or the data. The name *instance* may be familiar to those of you with programming backgrounds who understand class hierarchies. The document is an instance of the class defined by a DTD. If there is no DTD, the instance consists of an undefined class. If you don't have a programming background, think of a DTD as a set of rules. The document instance contains data described as a set that follows the rules set out in the DTD, which may or may not be embedded in the document. The document instance can be broken into a number of subcomponents, or entities. The entire root element is one such entity. Entities can be broken down further so

that individual elements and groups of elements can be considered entities, which you can, if you want, separate out of the document and link to the document instance using ENTITY declarations in your DTD.

Elements

Elements consist of three character-level components: the start tag, the end tag, and the content, if there is any. The names of elements must follow the production rules of XML names. Not all elements have content, and there is no rule that says they must.

More details of element construction can be found in Chapter 2, "Understanding and Creating Elements."

Start Tags Start tags begin with a <. This is called a left angle bracket. They must be followed by an XML name.

XML Names After creating the start tag, you name your element using an XML name. As we mentioned earlier, XML names may begin with an underscore, a letter, or a colon. They can contain letters, digits, periods, hyphens, underscores, colons, combining characters (a letter with a mark attached to it, such as an accent mark or a macron), and extending characters (which aren't letters but rather alphabetic symbols that some languages use and that act like letters in many ways).

XML names can never begin with numbers, periods, or hyphens.

WARNING

If you use a colon in an XML element—as in `<foo:Element/>`—most XML developers will immediately assume you are working with XML namespaces. General convention, but not the XML standard, demands that you reserve the colon for use with namespaces. And the standard itself is specific about what the intent of the colon is (namespace). Of course, the rebel in you may want to go against the grain of conventional thinking and use colons for purposes other than namespaces, but you'll run into enough resistance that you may decide to take up another cause. Most parsers will not validate a document that uses a colon in an XML name without a declared namespace.

End Tags End tags (also called closing tags) begin with a </ and end with a > . Their names must correspond exactly to the names of the beginning tags of the element description. If a start tag begins and an end tag with a different name of any kind follows, even if only the case is

different, no element exists, and the XML parser will generate an error. This is okay:

```
<Basic>Hello World!<anElement/></Basic>
```

The fragment `<anElement/>` is not an end tag and so does not violate the rule. It is a complete element tag set (it's an empty element with a start and closing tag). You can include a full element tag set within another element before closing an element tag. The full element tag set is said to be *nested* and is considered part of the element's content. The following will generate an error:

```
<Basic>Hello World!</basic>
```

Element Content The content is whatever lies between the start tag and the end tag. The following example is that of an element with content:

```
<foo:anotherEmpty>Data here</foo:anotherEmpty>
```

So is this:

```
<foo:anotherEmpty>Data here
<foo:empty/>
</foo:anotherEmpty>
```

Empty Elements An element does not have to have content. If you've seen an HTML `<hr>` element, you have seen an empty element at work. Translated to XML, the `<hr>` element would look like this: `<hr/>`. Even better would be the following, especially if you're worried about browser compatibility issues: `<hr />`.

If there is no content, the element is said to be an *empty element*. The following is an example of an empty element:

```
<empty/>
<foo:anotherEmpty></foo:anotherEmpty>
```

There is nothing wrong with writing out the entire start and end tag set of an empty element, although it does go against current convention somewhat, which tends to favor empty elements written as a single tag set rather than as a pair:

```
<foo:empty><foo:empty/>
```

The Root Element The document instance consists of the root element. Every other element must be contained within a root element. `Basic.xml` consisted of one element, which happened to be the root element. If we

add another element, that element *must* be contained within the root element:

```
<Basic>Hello World
<child>This is a child element</child>
</Basic>
```

In the preceding document fragment, the code highlighted in bold text consists of a child of the root element and the child's contents. The `child` element and its contents form the child of the root element.

When an XML processor encounters an element, it knows nothing about its semantics, which means it doesn't understand the *purpose* of the element. It doesn't care if you place a baby picture, or simply another element, within the `child` element to define it. You need to provide a way to give meaning to each element you create. Throughout this book you'll discover hundreds of ways to do this, which is one reason XML is so glorious.

Is It a Tag, or Is It an Element? Now that you've seen start and end tags and defined element content, it's useful to understand what a tag *isn't*. The following is not a tag:

```
<tagreference>This is not a tag</tagreference>
```

The preceding line of code represents an element, not a tag. A tag is a specific instance of markup that helps define an element. The preceding line consists of a start tag containing a generic identifier (`<tagreference>`), followed by some element content (`This is not a tag`), followed by an end tag (`</tagreference>`). Because we've determined it's not a tag, take another look at the preceding line of code and think about what it *is*. The combination of all the markup (the start tag and the end tag) and the element's content is the element. Think of a tag as a markup instance. Think of an element as the whole of all the parts—the tag, zero or more attributes, plus any content (if any content) in the element.

Building a Tree You can keep adding to the document instance until an entire *tree* of elements forms. XML trees consist of elements and element content, like a flow chart in descending order that begins with a parent element, which is the topmost element. The parent element in turn contains *child* elements. Child elements are so called because they are next in line on the descendant tree. These children may themselves have children, and so on. The tree is built from this hierarchical pattern. Some XML implementations, such as XSLT, consist of trees that consist of more than elements

and element content, which means that attributes and namespaces can be part of the tree. The DOM (which, as we explained earlier, is a set of interfaces for accessing XML trees programmatically) is another example of an implementation that uses a more granular approach to this definition of a tree.

For the purposes of Listing 1.5, however, let's keep the concept of a tree as simple as we can and limit it to elements and their content. Listing 1.6 later in this chapter shows a longer document that consists of several child elements that themselves consist of siblings.

Listing 1.5: Children and Siblings Create a Tree

```
<?xml version="1.0"?>
<Basic>Hello World
   <child type="siblingOfChildElement">
This is a child element</child>
   <child
 type="siblingOfChildElement">
This is another child element</child>
   <child type="siblingOfChildElement">
      <grandChild>
      This is a grandchild of the root element
         <greatGrandChild>This is a great
         grandchild of the root element
         </greatGrandChild>
      </grandChild>
   </child>
</Basic>
```

You can see we've discarded one part of our prolog—the XSL style sheet processing instruction. That way we can open the document in IE 5 and see the tree. By clicking the + and - symbols on the browser document window in IE 5, we can collapse and expand the tree. This is not a functionality of XML. This is done by a combination of CSS and Dynamic HTML within the scope of a default rendering object inherent to all instances of IE 5. By creating your own style sheet, you override this default mechanism in IE 5.

Figure 1.3 shows a truncated, but still well formed, version of this file before the tree is expanded. Figure 1.4 shows the same file with the tree expanded when the user clicks the + symbol. It uses empty elements to represent the child elements. You can see how XML's containerlike

methodology works and how the different elements branch off to make a tree-like structure.

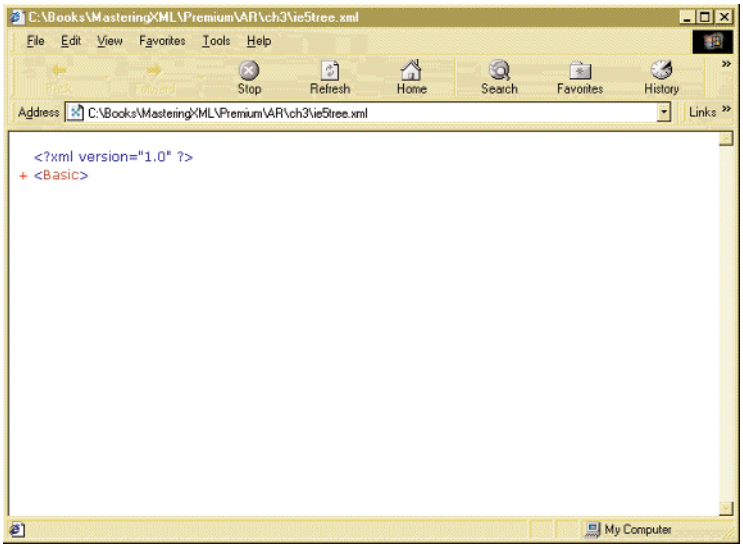

FIGURE 1.3: The tree collapsed in Internet Explorer 5

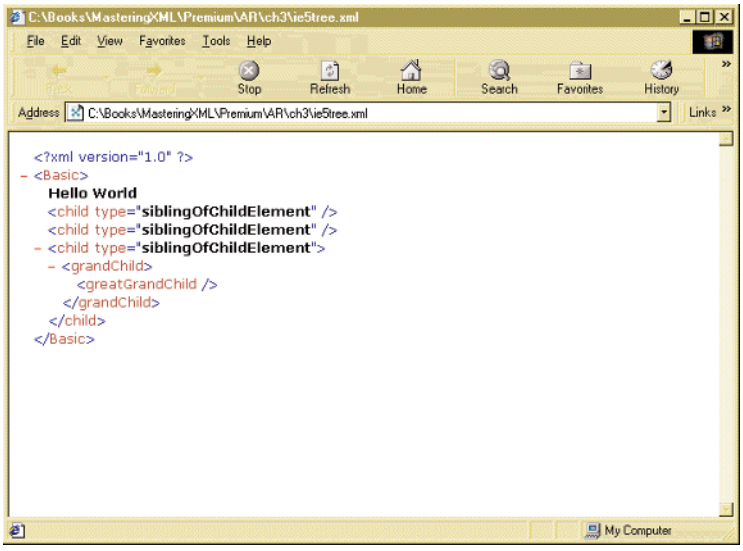

FIGURE 1.4 The tree expanded in Internet Explorer 5

Accessing the Document Tree In the world of object-oriented programming, everything in an XML document is a node. Start tags, end tags, empty element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions are all nodes, as is any one contiguous string of character data. Even the document itself is a node, which contains all the other nodes.

We won't go too in-depth here about all this, because plenty of space in this book is devoted to some XML-related programming concepts. It's worth pointing out, though, that a document tree can be accessed programmatically in one of two ways: through an event-based system or by the process of indexing the tree into memory and accessing the hierarchy. An event-based process, such as SAX, considers each node instance an event.

You can also navigate the tree of an XML document programmatically using the DOM. Each object of the hierarchy is indexed in memory and accessed by an object-oriented program, such as Java, or through JavaScript or another scripting language.

Is It Well Formed? Examine the following lines of code to see if you can identify which of these elements are well formed and which aren't. We'll identify the correct answers by referencing the code's line numbers in the paragraphs that follow.

```
1.  <fragment ><foo/>"Excellent," said Holmes. "That's why
    she cried, 'Liar!'"</fragment>
2.  <fragment foo="foo1" >
    <foo></foo><foo/>
    </fragment>
3.  <fragment><fragchild></fragment>
4.  <fragment><fragchild/></fragment>
5.  <fragment><fragchild<frag>/></fragment>
6.  <fragment><fragchild/>frag></fragment>
7.  <fragment><fragchild/>frag&gt;</fragment>
8.  <fragment><fragchild/>frag<</fragment>
9.  <fragment><fragchild/>
       <!– comment –>frag
    </fragment>
10. <fragment><fragchild/><STOP!/></fragment>
```

```
11.  <fragment><fragchild/><STOP?/></fragment>
```

Line 1 is okay. You can include character data with other elements in element content if you want, as long as you're not validating against a DTD that prohibits this.

Line 2 is also well formed. It contains two empty elements within a parent container.

Line 3 would be okay if the author had closed the `fragchild` element.

Line 4 is how Line 3 should look.

Line 5 is wrong. It contains a start tag before the end of the XML name. A parser will complain about an invalid XML name.

You might get away with Line 6, but you should always escape > characters, just to be safe.

Line 7 is better.

Line 8 is guaranteed to generate an error, because the < character will always be interpreted as a start tag by an XML processor.

Line 9 is well formed. It contains an element with content and a comment.

Lines 10 and 11 are not well formed. They each contain an invalid character in the element's name.

Attributes

Attributes are like modifiers. They describe certain properties of elements. They consist of attribute value pairs—the name of the attribute, followed by an equal sign, followed by the value of the attribute. The attribute must have a value:

```
<fragment foo="foo1" >
<foo/>"Excellent," said Holmes.
"That's why she cried, 'Liar!'"
</fragment>
```

Chapter 3, "Understanding and Creating Attributes," explores attribute notation in detail.

Examine the following lines of code to see if you can identify which of these attributes are well formed and which aren't. We'll identify the correct answers by referencing the code's line numbers in the subsequent list.

```
1.  <foo myatt="Pete's Place">
2.  <foo myatt="Pete's' Place">
```


```
3.  <foo test="'false'">
4.  <foo myatt='false">
5.  <foo myatt=false>
6.  <foo myatt='false'>
7.  <foo myatt="false">
8.  <AttachExpression
    Expression="&lt;A HREF=orderView3.asp?Order_ID=
➥[[Order_ID]]&gt;{{Order_ID}}&lt;/A&gt;"/>
9.  <FormatDate Format="%b %d, %Y"/>
```

Line 1 won't generate an error as long as the attribute type defined in an associated DTD is not a tokenized type even though there is a single quote nested within the double quotes. You should, however, use an apostrophe entity to represent the apostrophe (`&apos;`) here to avoid confusing the parser.

Line 2 is acceptable (though probably not the author's intent) and is an example of how a typo could actually be interpreted by the parser as a value.

Line 3 is also correct and demonstrates how nesting a string within a value can prove useful when trying to differentiate between a string value and some predefined data typed-value test.

Line 4 is a fairly obvious; we opened the attribute value with a single quote but closed with a double quote, so the statement will generate an error.

Line 5 is also easy because we left off the quotes, so an error will be generated.

Line 6 is fine, because we started with a single quote and ended with a single quote.

Line 7 is also fine, because we started with a double quote and ended with a double quote.

Line 8 also works, despite the heavy use of special characters, since none of them violate the rules that govern the use of attribute value types. This last one was tricky. You'll learn more about attribute value types in Chapter 3, but this is a good example of how you can combine XML with programming techniques to handle parameter passing.

Line 9 is well formed. The characters within the attribute values are CDATA, which allows for a wide range of values.

CDATA Sections

A CDATA section is an especially wonderful little markup unit that programmers will turn to often as their salvation for dealing with operators that conflict with XML rules. CDATA sections contain nothing but character data, no matter what their contents look like. They can contain the < and the & literal values. This means if you use them in a CDATA section, you don't need to escape them. A CDATA section has the funkiest syntax in all of XML, but it's worth the trouble.

If you're a JavaScript or Java developer, you can place all your JavaScript in a CDATA section, as is, without worrying about escaping the < and the & characters. The syntax for CDATA sections looks like this:

```
<![CDATA[content here]]>
```

A typical example, using Java, might look something like this:

```
<?xml version = "1.0"
encoding="UTF-8" standalone="yes" ?>
<fragment>
   <![CDATA[
   (while i <= 8)
   sum += i++;
   )]]>
</fragment>
```

Using CDATA sections provides clear advantages, and it's a good idea to use them whenever there's even a threat of a character that might be construed as XML markup by a processor.

Processing Instructions

Processing instructions consist of the following, in the exact order of their appearance in this list:

1. The string <?
2. The name of the processing instruction target, which can be any XML name unless the string is `XML` in either uppercase or lowercase (which would confuse the parser)
3. Optional white space characters

4. Any additional, and optional, characters (this is fairly open-ended because you might be passing parameter lists off to an external processor, although you can't use ?>, because the processor will think you're closing the instruction statement)

Remember that an XML declaration is not a processing instruction. An example of processing instructions is our reference to a style sheet processor at the beginning of this chapter. It bears repeating here:

```
<?xml-stylesheet type="text/xsl" href="basic.xsl"?>
```

You can use processing instructions for a variety of reasons, most of which have to do with extending the reach of the XML parser that is controlling the XML document you are working with. One simple example is for letting Web spiders know whether or not you want a page indexed:

```
<?robots index="no" follow="yes"?>
```

An XML processor itself does not do anything with instructions other than hand them to other processors.

Comments

Comments, like processing instructions, can appear in either the prolog or the document instance. Comments can't be read by machines—they are for people, so anything contained within a comment will not be parsed, including elements.

Comments are a good way to debug code. Debugging, or finding out what is wrong with your code, can be an art in any computer language, and comments have long been a way to isolate potential problem code chunks.

You need to keep a few rules in mind when using comments. If you've used them in HTML, they'll seem intuitive. If not, they're still rather simple:

- Comments begin with the string <!-- and must always close with this string: -->.
- The parser ignores everything between the <!-- and -->.
- Nothing can precede an XML declaration in an XML document, and comments are no exception.
- As in HTML, comments cannot be nested within a tag that defines an element name. The following will generate an error: `<tag <!-- this is not a well-formed comment -->`.

- Once you've started a comment, you can't use the -- characters together until you've decided to close your comment. Additionally, this means there is no such thing as a comment within a comment.

Introduction to Namespaces

If you build an XML document and define an element named `sound`, what happens if it needs to interact with another document that contains an element named `sound`, but has a meaning that is different from the same element in your document? You have a *collision* of elements. You need to find a means of dealing with the different ways elements work together when they have the same names. After all, you can't run around the world trying to make sure nobody uses *your* element name with a different meaning.

The answer is namespaces. You can create an attribute, either global or local in scope, that uniquely identifies your element through a URI. A URI is not the same as a URL (Uniform Resource Locator). A URI is a string that identifies a Web resource. It doesn't necessarily point to anything, even though it can.

NOTE

If you're a regular reader of the XML-Dev list, you'll know that there is a vocal contingent that would say, "Namespaces are not the answer!" However, namespaces are a part of XML today and have taken on considerable significance.

By identifying a namespace, you can create elements that are unique to that namespace and thus will be sure to have the meaning you intended.

Namespaces are created via an attribute that describes the namespace within which an element's definition resides. When a namespace is declared, a prefix is associated with each element bound to that namespace. The namespace is itself bound to a URI. A processing application that understands the set of rules for that namespace can then be used to process the data according to the rules set forth within the scope of that particular namespace. Let's look at a hypothetical example to explain how a namespace works.

A Hypothetical Namespace Application

Let's say you want to create a special set of elements that describe some specific functions of your company, a financial institution. To keep this simple, let's further say that there is only one element, called `bankrupt`. Well, bankrupt can mean different things to different people. There's out-of-money bankrupt. There's also morally bankrupt. But even within the scope of financial bankruptcy, there's a significant difference between the kind of bankruptcy most people experience and the kind a big company experiences. Further, some institutions may have their own specific definitions of bankruptcy, at least in the eyes of determining creditworthiness.

Therefore, your IT department has decided to take the bold step of defining its own internal vocabulary and binding it to a namespace. To do this, you've made your URI as unique as you think it can be (to avoid colliding with other similar vocabularies) by using your company's domain name as the binding entity. So, assuming your company is named Top Company, your URI for the application's namespace might be `http://www.topcompany.com/bankrupt/2001`. This URI does not necessarily point to anything. It merely acts as an identifier. Next, you need to be sure that your XML parser can understand that namespace. Unfortunately, since the namespace is created by your company, you'll probably need to build the processor yourself, because your XML parser *can't* understand the elements defined by that namespace.

So what does a namespace look like? First, you need to declare the namespace itself. This is accomplished by attaching an `xmlns` attribute to the element or elements bound to the namespace:

```
<?xml version="1.0"?>
<bk:bankrupt
xmlns:bk = "http://www.topcompany.com/bankrupt/2001"
>
```

From that point on, you can use the namespace and its bound element any way you want within the scope of the definitions you choose. It doesn't matter that there is no physical presence of your rules on the site listed in the declared namespace attribute. What *does* matter is that the processor used to parse those elements *understands* what those elements mean. Therefore, someone has to build the processor.

XSL and XSLT are two common namespace-driven XML lingoes. Unfortunately, XSL and XSLT stumbled out of the gate a bit when

Microsoft hurried out an XSL processor that used a namespace that was bound to an early, pre-standard version of the language. When the language became standardized, the namespace URI changed (it had to, because its rules changed, and the new rules would have been in conflict with the old if the namespace was the same). This caused considerable hair pulling, but it was a classic example of namespace use and the importance of understanding its use.

Namespaces can be scoped across more than one element, of course, and even across entire documents.

WHAT'S A NAMESPACE?

An XML namespace is a way of "qualifying" a set of elements and attributes. It's a way of mixing elements from multiple DTDs, or multiple sets of names, in a single document, and of saying which elements and attributes came from which set of names.

XML namespaces are used for three main reasons: intermixing vocabularies, intermixing document fragments, and establishing reserved names.

You use a namespace declaration to associate a URI reference with one or more XML elements, as a prefix to disambiguate them from each other. Then if two elements have the same name but different URI prefixes, they are considered to be different.

For example, consider this document fragment:

```
<myElement
   xmlns:foo="http://www.mydomain.com/foo.xsd"
   xmlns:fooFo="http://www.mydomain.com/fooFo.xsd"
>
<p>This is a p element that denotes a paragraph</p>
<foo:p>This is a "foo:p" element.
The "foo:p" element has a completely different pur-
pose than the "p" element.
</foo:p>
<fooFo:p>And this element has yet another completely
different purpose than either the "p" element or the
"foo" element.</fooFo:p>
```

We may need different definitions for what amounts to the same element name, p. So we append a prefix to each p element carrying a different meaning than the original, define what that prefix means by referencing it to a schema (which carries all the rules associated with that prefix), then append that same prefix when using it in the

CONTINUED ➞

document instance. So, according to a fictional `foo` schema referenced by the first namespace declaration in bold, the `foo:p` element does not represent a paragraph, but, perhaps, a proposal. It all depends on what the schema's intent is. You hope it's a well-defined schema.

You'll also see namespaces used in major XML vocabularies, like XSLT. The key in that case is that the processor must have instructions on how to handle the namespace. The URIs being pointed to don't necessarily exist in some physical space, like a URL does. A URI is an identifier, identifying for a processor which vocabulary is being used and which version of that vocabulary is being used. A processor either understands this vocabulary or it doesn't. If it doesn't, the element remains essentially meaningless.

Choosing between a DTD and a Schema

Up to this point we've focused our attention on creating XML documents based on DTDs. However, a significant portion of this book is devoted to another validation scheme for XML documents: *schemas*. Schemas, like DTDs, define the structure and semantics of an XML document, but in a more verbose way, using XML to define the rules and allowing for a richer set of data types to do so.

Schemas are an important enough development in XML that we've devoted two chapters to the topic. The first, Chapter 7, "An Introduction to Schemas," is a basic introduction. The second, Chapter 8, "Writing XML Schemas," delves more comprehensively into how to create them.

Many in the XML community believe that the DTD will not survive as a rules mechanism. Although this is extremely unlikely, especially considering that the core standards used to develop the schema vocabularies themselves are written using DTDs, it does point to their increasing value. This value stems largely from the rich data typing they provide, which means their elements can be more easily mapped against existing databases.

A schema definition is created by following a set of rules defined by the W3C that specifies how schemas should be set up. Schemas are defined within the framework of XML. An XML file is created that

describes exactly how to define all the elements in a document instance that conforms to a specific schema. This is similar to the function of DTDs. The main differences are that DTDs have a special syntax that looks different from the kind of syntax used in document instances and that DTDs have limited data typing capabilities.

A portion of a schema is shown in Listing 1.6. This schema has been truncated (all but one element definition has been taken out) for space purposes. By looking at the elements, you can probably figure out what is going on to some degree. The sequence and purpose of each element is defined in the schema document. When validated against a processor that validates against schemas, the XML document instance is matched to the definitions created in the schema. The XML document instance doesn't need to be a file. It can be a streaming instance passed from one server to another server as a message (which, in XML, is still considered a document). There are no specific rules about the physical nature of the XML document instance when using schemas. The only thing that matters is whether the document instance is valid against that schema.

Listing 1.6: Using a Schema to Define an Element

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema">
   <element name="lillybook">
      <annotation>
         <documentation>documentation
         </documentation>
      </annotation>
      <complexType>
         <sequence>
            <element ref="Title"/>
            <element ref="Author"/>
            <element ref="Abstract"/>
            <element ref="Chapter"/>
         </sequence>
         <attribute name="ref"
          type="string" use="required"/>
          <attribute name="id" type="string"
           use="required"/>
      </complexType>
   </element>
<!– additional schema elements here –>
</schema>
```

Deciding whether to choose DTD or schema validation for your XML development depends on a number of factors (and remember, you are not required to use). Ask yourself a few questions:

- Who's using the XML document? If it's for a massive audience, a DTD is usually the answer. If it's a more specific group, and you have confidence that the applications available to your target can process schema-based XML, choose schemas.
- Are you extracting data from a database? Most database vendors are leaning heavily toward schemas because of their data typing capabilities. This means that you can define integers, and even dates, in your schemas, thereby binding your elements to stronger data types that more closely resemble the real data in your database.
- Are you exchanging information with partners? If you are, schemas may be a good answer for managing the dialogue between two or more organizations.

If you're interested in immediately pursuing the advantages schemas offer, read through Chapter 7, "An Introduction to Schemas," for a more comprehensive look at how they work.

Building a Complete XML Document

This chapter has carefully reviewed each component of an XML document. Now, it's time to put your newfound knowledge to the test by examining a complete document. The document in Listing 1.7 demonstrates how easy it is to make a simple XML document with a minimal amount of effort (and a minimal DTD). Your XML documents in production environments are sure to be considerably more complex. As you continue with this book, your ability to create and work with such documents will grow.

Listing 1.7: A Complete XML Document

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE lillybook SYSTEM "lillybookv3.dtd">
<lillybook id="FreedomsDream"
xmlns:xlink="http://www.w3.org/1999/xlink">
   <Title>Freedom's Dream</Title>
```

```
   <Author>
      <FirstName>Chuck</FirstName>
      <LastName>White</LastName>
   </Author>
   <Abstract copyright="2001, Chuck White">
      <Paragraph>Descriptive narration for
      interactive short story "Freedom's Dream"
      </Paragraph>
   </Abstract>
   <Chapter>
      <ChapterTitle>One</ChapterTitle>
         <score source="chapter_1Score"/>
            <Section classification="multimedia"
             level="one">
               <SectionTitle>PartOne</SectionTitle>
       <Paragraph>Had it been a dream, Antron Crimea's memory
of the clenched fist piercing the sky of a tumultuous, thun-
dering crowd would have been bearable solitude. As it was
though, the reality brought him to another place, to a dis-
tance only something like a dream could take him.</Paragraph>
         <Paragraph>"The crowd forgot everything," is how
Antron described the situation to his psychiatrist,
<characterLink id="chesapeake" xlink:title="Chesapeake Alert"
xlink:href="Chesapeake.xml"
xlink:label="ChesapeakeAlert">Chesapeake
Alert</characterLink>. Antron remembered the rhythm, the
pulse, everything. After all this time the energy of the
crowd still seemed to reverberate through his
head.</Paragraph>
         <Paragraph>Chesapeake Alert was nothing but a large
bulbous mass of jelly-like flesh; a brain plopped down on an
empty, expensive slice of carpet. And though he had no legit-
imate locomotive capabilities of his own, he was aware of the
movements of a billion others.</Paragraph>
         <Paragraph>Antron's hundred legs crawled around what
was left of the carpet in the kind of pace unknown to you or
me. His earlier confusion had long ago been dissolved by the
righteous events of what he had seen during the course of
events Billy Freedom had ignited</Paragraph>
         <Paragraph>"Sometimes betrayal is a necessity," said
Chesapeake. "Startling. And expensive. It must be weighed
carefully."</Paragraph>
      </Section>
```

```
    </Chapter>
</lillybook>
```

You can download the DTD for Listing 1.7 at `www.tumeric.net/projects/books/complete/support/ch01_toc.asp`. The file is named `lillybookv3.dtd`. At this point in your XML development, you should focus most on the structure of the document. Notice the way each element is nested within another and that there is one root element.

The document begins with an XML declaration. There is no processing instruction for this document, but if we wanted to develop a style sheet or transformation for it, we would want to add a processing instruction to handle it. Next comes the DTD, which is external. After that, the document instance is parsed, beginning with the root element, `lillybook`.

You will also notice a number of other important attributes about the document, such as the consistent case use among elements and the fact that all the attribute values appear in quotes without any exceptions.

TIP

If you are creating a common type of document, such as an online book, you should really look to see if someone else has already created a publicly available DTD before setting out to create your own like we have here.

As you examine Listing 1.7, try to take everything you've learned into account and see if you can identify some of our main points about how to create XML documents. As you progress throughout the book, any mysteries remaining about Listing 1.7 will gradually clear. Let yourself explore the basic syntax. There are many more mysteries ready to be disclosed to you as the next several chapters unfold.

SUMMARY

In this chapter we introduced some of the basic concepts and constructs of XML documents. You now have an understanding of all the syntax requirements you'll need to create your first XML document.

In the next chapter, we'll explore how to create elements, which are one of the core entities in all XML documents.