CHAPTER 2

XML Structure

Reading and understanding the W3C specifications can be a difficult and daunting task. This chapter explains XML structures in an easy-to-understand way. This information is based on the third edition of the WC3's XML 1.0 specification. I did not use the XML 1.1 specification as a basis for this chapter in order to ensure the greatest compatibility amongst parsers and applications. In other words, the XML 1.0 specification is compatible with XML 1.1, but the reverse is not true.

This chapter will cover the basics for understanding and building an XML document. It begins with some fundamental concepts of XML; using these concepts, I'll break down the structure of a document and explain the syntax for document composition. Once you have a basic understanding of document structure, I'll introduce additional features such as namespaces and IDs. By the end of this chapter, you should be armed with enough knowledge not only to build XML documents but also to at least understand some of the more complex documents you may encounter. Although I'll present some information about DTDs, Chapter 3 provides more in-depth coverage.

Introducing Characters

XML uses most of the characters within the Unicode character set. The specification actually refers to the ISO 10646 character set, but usually you will find these two used interchangeably, because the two character sets are kept in sync. Unicode, a 32-bit character set, provides a standard and universal character set by assigning a unique number to every character. This way, by using Unicode, data is the same without regard to language or country. The two Unicode formats, which all parsers must accept, are UTF-8 and UTF-16, although you can use other character encodings as long as they comply with Unicode.

Character References

Characters cannot always be represented in their literal formats. Also, sometimes certain characters in their literal forms are invalid to use because they violate the XML specification, which depends upon the type of markup being used at the time. Character references represent the literal forms using their numeric equivalents. You can express character references in two ways: using decimal notation or hexadecimal notation. For example:

- The character *A* in decimal format is `&#65;`.
- The character *A* in hexadecimal format is `&x41;`.

The only constraint for the character to be considered well-formed is that it conforms to the rules for valid characters, which are expressed in hexadecimal format and include the following range of characters:

```
#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

Whitespace

Throughout this chapter, you will encounter the term *whitespace*. Whitespace, as used within XML, consists of one or more of the following characters (expressed in hexadecimal): `#x20` (space), `#x9` (tab), `#xD` (carriage return), or `#xA` (line feed). By default, whitespace is significant within an XML document. In *most* cases, it is up to the application to determine how it wants to handle whitespace. As you will see later in this chapter in the section "Using xml:space and xml:lang," `xml:space` is a way to force an application to preserve whitespace.

Names

The term *name*, as used within this chapter for explaining XML syntax, defines the valid sequence of characters that you can use. A name begins with an alphabetical character, an underscore, or a colon and is followed by any combination of alphanumeric characters, periods, hyphens, underscores, and colons, as well as a few additional characters defined by `CombiningChar` and `Extender` within the XML specification.

Names beginning with the case-insensitive `xml` are also reserved by the current and future XML specifications. For example, names already in use include `xmlns` and `xml`. Basically, it is not wise to use a name beginning with those three letters. It is also not good practice to use colons in names. Although you will find people using them, especially when using the DOM and not using namespace-aware functionality, using colons can lead to problems when not used for namespace purposes. Table 2-1 shows some example names.

Table 2-1. *Example Names*

Valid Names	Invalid Names
`automobile1`	`1automobile`
`_automobile`	`+automobile`
`:automobile`	`(automobile`
`my.automobile`	`.automobile`
`my:_automobile`	`@automobile`

Character Data

Markup consists of XML declarations, document type declarations, elements, entity references, character references, comments, processing instructions (PIs), CDATA section delimiters, text declarations, and any whitespace outside the document element and not contained within other markup. An example of whitespace that is considered markup is the line feed used between the prolog and the body. Character data, simply, is everything else that is not markup. It is the actual content of the document, which is being described and structured by the markup.

A few characters require special attention:

- Less-than sign (<)
- Ampersand (&)
- Greater-than sign (>)
- Double quote (")
- Single quote (')

Except when used for markup delimiters or within a comment, PI, or CDATA section, & and < can never be used directly. The > character must never be used when creating a string containing]]> within content and not being used at that time to close a CDATA section. The double and single quote characters must never be used in literal form within an attribute value. Attribute values may be enclosed within either double or single quotes, so to avoid potential conflicts, those characters are not allowed within the value. All these characters, according to their particular rule sets, must be represented using either the numeric character references or the entity references, as shown in Table 2-2.

■ **Note** The entity references for these special characters do not need to be defined in a DTD because they are automatically built into the parser.

Table 2-2. *Special Character Representations*

Character	Character Reference (Decimal)	Character Reference (Hexadecimal)	Entity Reference
<	<	<	<
&	&	&	&
>	>	>	>
"	<	<	<
'	'	'	'

Case Sensitivity

XML is case-sensitive. You must be careful when writing markup to ensure that you use case correctly. An element that has a start tag in all lowercase must have an end tag that is also in all lowercase. This also is important to remember when using attributes. The attribute `a` is not the same as the attribute `A`. It is a good idea to be consistent with case within a document. All attributes should use the same case; lowercase is commonly used for attributes. Element names should also be consistent. The common methods for case in elements names are using all lowercase, using all uppercase, or using uppercase for the first letter of a word and using lowercase for the rest of the word. For example:

```
<document>
     <MyElement>content here</MyElement>
     <MYELEMENT>content here</MYELEMENT>
     <myelement a="1" b="2" />
     <!-- The following is well-formed,
          but it is not good to mix attribute cases -->
     <myelement a="1" A="2" />
     <!-- The following is invalid because of mismatching start and end tags -->
     <MYELEMENT>content here </myelement>
</document>
```

Understanding Basic Layout

An XML document describes content and must be well-formed, as defined in the WC3's XML specifications. The bare minimum for a well-formed document is a single element that is properly started and terminated. This element is called the *root* or *document element*. It serves as the container for any content. A document's layout consists of an optional prolog; a document body, which consists of the document element and everything it contains; and an optional epilog.

Prolog

A *prolog* provides information about the document. A prolog may consist of the following (in this order): an XML declaration; any number of comments, PIs, or whitespace; a document type declaration; and then again any number of comments, PIs, or whitespace. Though not required, an XML declaration is highly recommended. You can find information about comments and PIs in the section "Understanding Basic Syntax." Listing 2-1 shows an example prolog.

Listing 2-1. *Example Prolog*

```
<?xml version="1.0"?>
<!--The previous line contains the XML declaration -->
<!--The following document type declaration contains no subsets -->
<!DOCTYPE foo [
]>
<!--This is the end of the prolog -->
```

The prolog in Listing 2-1 takes the form of an XML declaration, two comments, a document type declaration, and another comment.

XML Declaration

The *XML declaration*, the first line in Listing 2-1, provides information about the version of the XML specification used for document construction, the encoding of the document, and whether the document is self-contained or requires an external DTD. The basic rules for composition of the declaration are that it must begin with `<?xml`, it must contain the version, and it must end with `?>`. Documents containing no XML declaration are treated as if the version

were specified as 1.0. When using an XML declaration, it must be the first line of the document. No whitespace is allowed before the XML declaration. Listing 2-2 shows an example XML declaration.

Listing 2-2. *Example XML Declaration*

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

Version

The version information (`version`), which is mandatory when using an XML declaration, indicates to which XML specification the document conforms. The major difference between the two specifications, XML 1.0 and XML 1.1, is the allowed characters. XML 1.1 allows flexibility and supports the changes to the Unicode standards. The rationale behind creating a new version rather than modifying the XML 1.0 specification was to avoid breaking existing XML parsers. Parsers that support XML 1.0 are not required to support XML 1.1, but those that support XML 1.1 *are* required to support XML 1.0. With respect to the XML declaration, the version either can be 1.0, as in `version="1.0"` (as shown in Listing 2-2), or can be 1.1, as in `version="1.1"`.

Encoding

The encoding declaration (`encoding`), which is not required in the XML declaration, indicates the character encoding used within the document. Encodings include, but are not limited to, UTF-8, UTF-16, ISO-8859-1, and ISO-2022-JP. It is recommended that the character sets used are ones registered with the Internet Assigned Numbers Authority (IANA). When encoding is omitted and not specified by other means, such as byte order mark (BOM) or external protocol, the XML document must use either UTF-8 or UTF-16 encoding. Although Listing 2-2 explicitly sets the encoding to UTF-8, this is not needed because UTF-8 is supported by default.

Stand-alone

The stand-alone declaration (`standalone`), also not required within the XML declaration, indicates whether the document requires outside resources, such as an external DTD. The value `yes` means the document is self-contained, and the value `no` indicates that external resources may be required. Documents that do not include a stand-alone declaration within the XML declaration, yet do include external resources, automatically assume the value of `no`.

Document Type Declaration

The *document type declaration* (`DOCTYPE`) provides the DTD for the document. It may include an internal subset, which means declarations would be declared directly within the `DOCTYPE`, and/or include an external subset, which means it could include declarations from an external source. The internal and external subsets collectively are the DTD for the document. Chapter 3 covers DTDs in detail. Listing 2-3, Listing 2-4, and Listing 2-5 show some example DTDs.

Listing 2-3. *Document Type Declaration with External Subset*

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

Listing 2-4. *Document Type Declaration with Internal Subset*

```
<!DOCTYPE foo [
     <!ELEMENT foo (#PCDATA)>
]>
```

Listing 2-5. *Document Type Declaration with Internal and External Subset*

```
<!DOCTYPE foo SYSTEM "foo.dtd" [
     <!ELEMENT foo (#PCDATA)>
]>
```

Body

The *body* of an XML document consists of the document element and its content. In the simplest case, the body can be a single, empty element. You may have heard the term *document tree* before; this term is synonymous with the body. The document element is the base of the tree and branches out through elements contained within the document element. The section "Understanding Basic Syntax" covers the basic building blocks of the body. Listing 2-6 shows an example of a document body.

Listing 2-6. *Example of an XML Document Body*

```
<root>
     <element1>Some Content</element1>
     <element2 attr1="attribute value">More Content</element2>
</root>
```

Epilog

If you are referring to the XML specifications, you will not find a reference to the *epilog*. Within the XML specifications, the epilog is equivalent to the `Misc*` portion of the document definition as defined using the Extended Backus-Naur Form (EBNF) notation. For example:

```
document   ::=   prolog element Misc*
```

The epilog refers to the markup following the close of the body. It can contain comments, PIs, and whitespace. Epilogs are not mandatory and, other than possibly containing whitespace, are not very common. Many parsers will not even parse past the closing tag of the document element. Because of this limitation, a possible use for the epilog is to add some comments for someone reading the XML document. This type of usage of an epilog causes no problems if a parser does not read it.

Understanding Basic Syntax

XML syntax is actually pretty simple. Many people get away with documents consisting of only elements and text content. These documents tend to have a simple structure with simple data, but isn't that the whole point of XML in the first place? Once you begin working with

more complex documents, such as those involving namespaces and content that is not just valid plain text, you may start to get a little intimidated. I know the first time I ever encountered a schema, I felt a little overwhelmed.

After reading the following sections, you should understand at least the basics of XML documents and be able to understand documents used in some XML techniques such as validation using schemas, SOAP, and RELAX NG. Some documents may seem impossible to ever understand, but armed with the basic knowledge in this chapter, you should be able to find your way.

Elements

Elements are the foundation of a document, and at least one is required for a well-formed document. An element consists of a start tag, an end tag, and content, which is everything between the start and end tags. Elements with no content are the exception to this rule because the element may consist of a single empty-element tag.

Start Tags

Start tags consist of <, the name, any number of attributes, and then >. *Name* refers to a valid, legal name as explained within the "Characters" section.

This shows an element start tag named `MyNode` having one attribute:

```
<MyNode att1="first attribute">
```

End Tags

End tags take the form of `</", Name, ">`, where `Name` is the same as the starting tag. The end tag for the previous example would be as follows:

```
</MyNode>
```

Element Content

Content may consist of character data, elements, references, CDATA sections, PIs, and comments. Everything contained within the element's start and end tags is considered to be an element's content. For example:

```
<myElement>
     <nestedElement>content of nestedElement</nestedElement>
</myElement>
```

Breaking this document down, the element name `nestedElement` contains a string of character data. The document element `myElement` contains content consisting of whitespace (a line feed and then a tab), followed the element `nestedElement` and its content, followed by more whitespace (line feed).

Empty-Element Tags

Elements without content can appear in the form of a start tag directly followed by an end tag (as well as without any whitespace). To simplify expressing this, you can use an *empty-element tag*. Empty-element tags take the form of `<", Name, "/>`. For example:

```
<!-- start and end tags without content -->
<myElement></myElement>

<!-- empty-element tag -->
<myElement/>

<!-- start and end tags WITH content -->
<myElement> </myElement>
```

Notice that the last example does contain content. Even though it's only a single space, the element contains content. Every character, including whitespace, is considered content.

Element Hierarchy

The most important point to remember when dealing with XML is that it must be well-formed. This may be redundant information, but if you are coming from the HTML world, it can be easy to forget. It's easy to get away with malformed documents when writing HTML, especially because not all tags are required to be closed. Take the HTML document shown in Listing 2-7, for example.

Listing 2-7. *HTML Example*

```
<HTML><BODY>
     <P>This is all in <I>Italics and this is <B>Bold</I></B><BR>
     New line here</P>
     <form name="myform" method="post" action="mypage.php">
        <table width="100%" border="0">
           <tr valign="top">
              <td>Name: <input type="text" name="name" value=""></td>
           </tr>
           <tr>
              <td><input type="submit" name="submit" value="Submit">
                 </form>
              </td>
           </tr>
        </table>
</BODY></HTML>
```

The document in Listing 2-7 is not well-formed at all. The simplest piece to identify is that the `BR` tag is opened and never closed. Within the `P` tag, the hierarchy is completely broken. Beginning with the `I` tag, you'll see some text followed by an opening `B` tag. Using XML rules, you would expect the `B` tag to be closed prior to the `I` tag, but as illustrated, the `I` tag is actually closed first and then the `B` tag is closed. If you have ever wondered why XML tends to be illustrated in an indented format, well, the answer might be much clearer now. Not only is the document easier for human readability, it also is easier to find problems in malformed documents.

The hierarchy of tags is completely invalid in Listing 2-7. Not only is there a problem with the `B` and `I` tags, but also the opening and closing form and table tags do not nest correctly. When writing HTML, it's all about presentation in the browser. A problem many UI designers

ran into years ago, before the days of CSS, was related to forms and tables. Depending upon the placement of the `form` and `table` tags, additional whitespace would appear in the rendered page within a Web browser. To remove the additional whitespace, designers would open forms prior to the `table` tag and close them before closing the table. Web browsers, being forgiving, would render the output correctly without the extra whitespace even though the syntax of the document was not actually correct. As far as XML is concerned, that type of document is not well-formed and will not parse. Elements must be properly nested, which means they must be opened and closed within the same scope. In Listing 2-7, the `table` tag is opened within the scope of the `form` tag but closed after the `form` tag has been closed. Even though it may render when viewed in a browser, the structure is broken and flawed because the `form` tag should not be closed until all tags residing within its scope have been properly terminated.

Each time an element tag (start, end, or empty element) is encountered, you should insert a line feed and a certain number of indents. Typically for each level of the tree you descend (each time you encounter an element start tag), you should indent one more time than you did the previous time. When ascending the tree (each time an element's end tag is encountered), you should index one less time than previously. Because an empty-element tag serves both purposes, it can be ignored. If you tried to do this with the example from Listing 2-7, you just could not do it. Using whitespace for formatting also makes it pretty easy to spot where it is broken as well:

```
<HTML>
     <BODY>
          <P>This is in
               <I>Italics and this is
                    <B>Bold
                    </I>
               </B>
               <BR>New Line here
          </P>
          <form name="myform" method="post" action="mypage.php">
               <table width="100%" border="0">
                    <tr valign="top">
                         <td>Name:
                              <input type="text" name="name" value="">
                              </td>
                         </tr>
                         <tr>
                              <td>
                                  <input type="submit" name="submit" value="Submit">
                                  </form>
                              </td>
                         </tr>
                    </table>
     </BODY>
</HTML>
```

Although this document has several issues, the most obvious problem should jump out at you. The indenting is completely off between the closing `table` tag and the closing `BODY` tag.

This clearly indicates something is wrong with the document. The document in Listing 2-8 applies the rules for XML elements to the document from Listing 2-7 to produce a well-formed XML document.

Listing 2-8. *HTML Example Using Well-Formed XML*

```
<HTML>
     <BODY>
          <P>This is in
               <I>Italics and this is
                    <B>Bold</B>
               </I>
               <BR/>
          </P>
          <form name="myform" method="post" action="mypage.php">
               <table width="100%" border="0">
                    <tr valign="top">
                         <td>Name:
                              <input type="text" name="name" value="" />
                         </td>
                    </tr>
                    <tr>
                         <td>
                              <input type="submit" name="submit" value="Submit" />
                         </td>
                    </tr>
               </table>
          </form>
     </BODY>
</HTML>
```

This might also give you an inclination of why Extensible HTML (XHTML) was created. XHTML is a stricter version of HTML that not only can be processed by a browser but, because it is XML compliant, can also be processed by applications.

Attributes

You can think of *attributes* as properties of an element, similar to properties of an object. You might be shaking your head right now completely disagreeing with me. You are 100 percent correct, but for a simple document and to give at least a basic idea of what they are, I will use that analogy for now. Attributes can exist within element start tags and empty-element tags. In no case may they appear in an element end tag. Attributes take the form of name/value pairs using the following syntax: `Name="Value"` or `Name='Value'`. You can surround values with either double or single quotes. However, you must use the same type of quotes to encapsulate the attribute's value. It also is perfectly acceptable to use one style of quotes for one attribute and another style for a different attribute. The attribute name must conform to the constraints defined by the term *name* earlier in this chapter. Also, attributes

within an element must be uniquely named, meaning an element cannot contain more than one attribute with the same name. Listing 2-9 shows an invalid attribute usage.

Listing 2-9. *Invalid Attribute Usage*

```
</Car color="black">
<Car color="black" color='white' />
```

Attributes also have no specified order within the element, so the following two examples are identical, even though the order and quoting are different:

```
<Car make="Ford" color="black" />
<Car color="black" make='Ford' />
```

Attribute Values

Attributes must also have a *value*, even if the value is empty. Again, referring to HTML, you may be accustomed to seeing lone attribute names such as `<HR size="5" noshade>` or `<frame name="xxx" scrolling="NO" noresize>`. Notice that `noshade` and `noresize` have no defined values. These are not well-formed XML and to be made conformant must be written as `<HR size="5" noshade="noshade">` and `<frame name="xxx" scrolling="NO" noresize="noresize">`, which now makes them XHTML and XML compliant. In cases where an attribute value is empty and there are no rules for any default values, such as those for converting HTML to XHTML, you would write an attribute as such: `attrname=""`.

Attribute values can also not contain unescaped `<` or `&` characters. Also, you should use escaped characters for double and single quotes. Although it might be OK to use a literal single quote character within an attribute value that is encapsulated by double quotes (though in this case double quote characters must be escaped), it is not good practice and highly discouraged.

Suppose you wanted to add some attributes to the element `Car` with the following name/value pairs:

- `color`: Black and white
- `owner`: Rob's
- `score`: Less than 5

You would write this as follows:

```
<Car color="black &amp; white" owner="Rob&apos;s" score="&lt; 5" />
```

Attribute Use

The use of attributes, unless specifically required such as through a DTD, is really a choice left to the document author. You will find opinions on attribute use running the full spectrum, with some saying you should never use attributes. When considering whether you should use an attribute or whether it should be a child element, you have a few facts to consider. If you can answer "yes" to any of the following questions, then you should use an element rather than an attribute:

- Could multiple values apply to an element?
- Is a DTD requiring the attribute being used?
- Is the data essential to the document and not just an instruction for an application?
- Is the value complex data or difficult to understand?
- Does the value need to be extensible for potential future use?

If the questions aren't applicable, then it comes down to personal preference. One point to always remember is that the document should end up being easily understood by a human and not just meant for electronic processing. With this in mind, you have to ask yourself which of the following is easier to understand. This is the first choice:

```
<Car make='Ford' color='black' year='1990' model='Escort' />
```

and this is the second choice:

```
<Car>
     <make>Ford</make>
     <color>black</color>
     <year>1990</year>
     <model>Escort</model>
</Car>
```

CDATA

CDATA sections allow the use of all valid Unicode characters in their literal forms. The CDATA contents bypass parsing so are great to use when trying to include content containing markup that should be taken in its literal form and not processed as part of the document. CDATA sections begin with `<![CDATA[`, which is followed by your content, and end with `]]>`, like so:

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

The only invalid content in this example is the literal string `]]>`. As you may have guessed, using `]]>` indicates the close of the CDATA section. To represent this string, you would need to use `]]>`.

For example, if you were writing an article about using XML and were using XML as the document structure, CDATA sections would allow you to embed your examples without requiring any character escaping. Listing 2-10 shows an example without a CDATA section, and Listing 2-11 shows an example with one.

Listing 2-10. *Example Without a CDATA Section*

```
<document>
     <title>Example of an XML</title>
     <example>
          &lt;xml version="1.0"?&gt;
          &lt;document&gt;
          this &amp;amp; that
          &lt;/document&gt;
     </example>
</document>
```

Listing 2-11. *Example Using CDATA Section*

```
<document>
     <title>Example of an XML</title>
     <example><![CDATA[
          <xml version="1.0">
          <document>
               this & that
          </document>
     ]]></example>
</document>
```

Clearly, the document in Listing 2-11 is much easier to read than the one in Listing 2-10. If editing a document by hand, it is also easier to write because you don't need to be concerned with figuring out what the correct entities to use are.

Because of the flexibility of CDATA sections, you may have heard or read somewhere that CDATA is great to use for binary data. In its native form, this is not true. You have no guarantee that the binary data will not contain the characters `]]>`. For this reason, binary data that must be encoded should use a format such as Base64. Now, if Base64 is used for encoding, a CDATA section is not even necessary, and it could be embedded directly as an element's content. This is because Base64 does not use any of the characters that would be deemed illegal for element content.

Comments

You can use *comments* to add notes to a document. This is comparable to a developer adding comments to source code. They do not affect the document but can be used to add some notes or information for someone reading it. For this reason, parsers are not required to parse comments, although most will allow access to the content. This is what a comment looks like:

```
<!-- This is a comment -->
```

Comments consist of the initial `<!--`, the actual text for the comment, and finally the closing `-->`. Be aware of the following stipulations when using comments:

- The content for a comment must not contain `--`.
- A comment may not end with `-`.

Other than those conditions, comments can contain any other characters.

Comments may also occur anywhere after the XML declaration as long as they are not contained within markup. Listing 2-12 shows some valid comments, and Listing 2-13 shows some invalid ones.

Listing 2-12. *Valid Comments*

```
<!-- The <Car> elements do not contain all known automobiles -->
<!-- This is valid as a whitespace follows the last "-" character - -->
<!-- Don't forget to escape the & character when used as element content -->
```

Listing 2-13. *Invalid Comments*

```
<!-- Comments take the form of <!-- This is a comment --> within a document -->
<!-- This comment is invalid as it ends with three "-" characters. --->
<Car <!-- Invalid because it resides within the element start tag -->>
```

Processing Instructions

XML is purely concerned with document content. A PI allows application-specific instructions to be passed with the document to indicate to the application how it should be processed. The PI takes the form of <?, which is followed by the target (which must be a valid name) and white-space, then takes the actual instruction, and closes with ?>, like so:

```
<?target instructions ?>
```

The target indicates the application that the instruction targets. You might already be familiar with this syntax from PHP:

```
<?php echo "Hello World"; ?>
```

This syntax is a PI. The PI target is php, and the instruction is echo "Hello World";. If you were creating an XHTML document and embedding PHP code, this would constitute a well-formed XML document.

Another case you may have already encountered is the association of style sheets with an XML document. Many XML editors will add the following PI so they can easily perform XSL transformations on the XML you may be editing:

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

Entity References

You have already encountered some of the built-in entity references (&, <, >, ', and ") throughout this chapter. Just as characters can be represented using numeric character references, entity references are used to reference strings, which are defined in the DTD. They take the form of &, which is followed by a legal name, and they terminate with a semicolon. You are probably familiar with the concept from HTML:

```
<P> Copyright &copy; 2002</P>
```

The entity reference © is defined in the HTML DTD and represents the copyright symbol. Entity references cannot just be used blindly, however. The document must provide a meaning to an entity reference. For instance, if you were looking at a document that contained <p>&myref;<p>, the entity reference &myref; has absolutely no meaning to you or may mean something completely different to you than to me. You can use DTDs to define an entity reference. It is mandatory that any entity reference, other than those that are built in, must be defined. Looking at an HTML page, you may notice the DOCTYPE tag at the top of the page. The contents depend upon the type of HTML you are writing. For instance, -//W3C//DTD HTML 4.01 Transitional//EN refers to the DTD http://www.w3.org/TR/html4/loose.dtd. This file contains a reference to http://www.w3.org/TR/html4/HTMLlat1.ent. If you looked at the contents of this file, you will notice that the entity copy is defined as <!ENTITY copy CDATA © -- copyright sign, U+00A9 ISOnum -->.

The entity reference, when used within the document, then is able to take its "meaning" from the definition. This is further explained in Chapter 3.

General Entity Declaration

Entity declarations may be either general or parameter entity declarations. Entity declarations will be covered in more depth in Chapter 3, though general entities have some bearing to this discussion with respect to entity references. The common use of general entities is to declare the text replacement value for entity references. General entities are commonly referred to as *entities* unless used in a context where that name would be ambiguous; therefore, for the sake of this section, *entities* will refer to general entities.

Entities are defined within the DTD, which is part of the prolog. Suppose you had the string `"This is replacement text"`, which you want to use many times within the document. You could create an entity with a legal name, in this case `"replaceit"`:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
     <!ENTITY replaceit "This is replacement text">
]>
<foo>&replaceit;</foo>
```

If this document were loaded into a parser that was substituting entities, which means it is replacing the entity reference (`&replaceit;`) with the text string defined in the entity declaration, the results would look something like this:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
     <!ENTITY replaceit "This is replacement text">
]>
<foo>This is replacement text</foo>
```

Using Namespaces

Documents can become quite complex. They can consist of your own XML as well as XML from outside sources. Element and attribute names can start overlapping, which then makes the names ambiguous. How do you determine whether the name comes from your data or from an outside source? Looking at the document, you would have to guess what the elements and attributes mean depending on the context. Unfortunately, applications processing the XML typically don't understand context, so the document would no longer have the correct meaning. *Namespaces* solve this potential problem.

Namespaces are collections of names identified by URIs. They are not part of the XML specification but have their own specification that applies to XML. Through the use of namespaces, names within a document are able to retain their original meanings even when combined with another document that contains some of the same names with completely different meanings.

Assume you are building a document that includes customer information as well as items they have ordered, and assume your customer records look like the following:

```
<Customer>
     <Name>John Smith</Name>
     <Number>12345</Number>
</Customer>
```

The items ordered by the customer take the form of the following structure:

```
<Items>
     <Item>
          <Name>Book</Name>
          <Number>11111</Number>
     </Item>
</Items>
```

Combining these into a single document would result in the following:

```
<Order>
     <Customer>
          <Name>John Smith</Name>
          <Number>12345</Number>
     </Customer>
     <Items>
          <Item>
               <Name>Book</Name>
               <Number>11111</Number>
          </Item>
     </Items>
</Order>
```

Unless you read the pieces of the document in context, the elements `Name` and `Number` are ambiguous. Does `Number` refer to the customer number or an item number? Right now the only way you can tell is that if you are within an item, then `Number` must refer to an item number; otherwise, it refers to a customer number. This is just a simple case, but it does get worse, such as when elements appear within the same scope. In any event, using namespaces uniquely identifies the elements and attributes, so there is no need for guesswork or trying to figure out the context. Take the following document, for instance. Separate namespaces have been created for `Customer` and `Item` data. Just by looking at the element names, you can easily distinguish to what the data refers.

```
<Order xmlns:cus="http://www.example.com/Customer"
       xmlns:item="http://www.example.com/Item">
     <cus:Customer>
     <cus:Name>John Smith</cus:Name>
          <cus:Number>12345</cus:Number>
     </cus:Customer>
     <item:Items>
          <item:Item>
               <item:Name>Book</item:Name>
               <item:Number>11111</item:Number>
          </item:Item>
     </item:Items>
</Order>
```

Defining Namespaces

Looking at the previous example, you may have already determined that `xmlns:cus="http://www.example.com/Customer"` is a namespace definition. Usually, and I stress *usually*, this is not the case; namespaces are created using a special prefixed attribute name and a URI, like so:

```
xmlns:prefix="URI"
```

Based on this definition, `prefix` refers to the namespace prefix you want to use throughout your document to associate certain elements and attributes to a namespace name (`URI`). In this example, the `Number` element within the `Customer` element becomes `cus:Number`, and the `Number` element within the `Item` element becomes `item:Number`. Now, the XML clearly distinguishes between the meanings of these two elements. You have removed any ambiguity from the document.

These new names being used in the elements are called *qualified names*, also referred to as *QNames*. They can be broken down into two parts, separated by a colon: the prefix and the local name. When using namespaced elements, the start and end tags now must contain the qualified name. Again, an exception to this exists, which you will come to in the "Default Namespace" section.

The significant portion of the namespace declaration is the URI (the namespace name). Once bound to a node or element, this will never change. The prefix, however, is not guaranteed. By manipulating the tree, such as moving elements around using the DOM, it is possible a namespace collision may occur. This frequently happens when a namespace defined lower in the tree declares a namespace and uses a prefix, which was used in one of its ancestors. By moving some element as a child of this other element, the prefixes would collide because they refer to two different URIs. It is perfectly valid for the prefix to automatically be changed to one that would not conflict. This is covered in more detail in the section "Namespace Scope."

Elements containing the namespace definition are not part of the namespace unless prefixed. Listing 2-14 shows the `Order` element within a namespace, because it is prefixed with `ord`, as specified in the namespace definition. The `Order` element in Listing 2-15 is not in any namespace even though a namespace is being defined.

Listing 2-14. *Element Order Within the* `http://www.example.com/Order` *Namespace*

```
<ord:Order xmlns:ord="http://www.example.com/Order" />
```

Listing 2-15. *Element Order Not Within the* `http://www.example.com/Order` *Namespace*

```
<Order xmlns:ord="http://www.example.com/Order" />
```

Namespaces are not required for every element and attribute within a document. You need to remember that namespaces remove ambiguity when there are, or there could be, overlapping names. Looking at the example, the only two elements that require namespacing are `Name` and `Number`. It would have been perfectly valid to not put all other elements into namespaces.

Namespaces can also apply to attributes as well:

```
<cus:Customer cus:cid="12345" />
```

The attribute `cid`, with the `cus` prefix, falls within the `http://www.example.com/Customer` namespace.

Default Namespaces

All rules have exceptions. If you remember from the previous section that namespaces take the form of `prefix:name`, well here is the exception: default namespaces allow a namespace to be defined that causes all elements, unless explicitly set to a namespace, to automatically be assigned to the default namespace, like so:

```
<Order xmlns="http://www.example.com/Order" />
```

You may think that the `Order` element is not associated with any namespace. This, however, is wrong. Default namespaces apply to the element they are defined on as well as to all elements, but not to attributes contained in the defining element, unless already associated with a namespace using the QName approach.

■ **Caution** Default namespaces do not affect attributes. Unless explicitly set to a namespace with a prefix, attributes do not belong to any namespace. This is extremely important to remember when working with many of the XML technologies, not just the ones within PHP. This knowledge may save you many hours and days of trying to debug an XML-based project.

Let's return to a simplified version of the order structure:

```
<Order xmlns="http://www.example.com/Order"
       xmlns:item="http://www.example.com/Item">
     <Items>
          <Item itid="12345">
               <item:Name>Book</item:Name>
               <item:Number>11111</item:Number>
          </Item>
     </Items>
</Order>
```

This structure contains two namespaces. One is `http://www.example.com/Item`, which is referenced by the prefix `item`, and the other, `http://www.example.com/Order`, is a default namespace. Based on the structure, the elements `Name` and `Number` belong to the `http://www.example.com/Item` namespace because they are using QNames with the `item` prefix. The elements `Order`, `Items`, and `Item` all belong to the `http://www.example.com/Order` namespace, because they are not explicitly set to any namespace so inherit the default namespace. Lastly, the attribute `itid` does not belong to any namespace. It is not explicitly set and hence doesn't use a QName, and as you remember, attributes do not inherit the default namespace.

If possible, I recommend avoiding default namespaces and using QNames with namespaces. As documents become more complex, they become much more difficult to read and understand. Default namespaces do not easily stand out, and when adding namespace scope to the equation, they can become quite confusing to follow. Using qualified names also will help avoid the confusion that sometimes happens with attributes; many people have been bitten by the fact that attributes do not inherit the default namespace and have spent a great deal of time trying to find the bugs in their XML.

Reserved Prefixes and Namespace Names

By default, XML processors are required to define two namespaces with associated prefixes by default:

- The prefix `xml` is bound to `http://www.w3.org/XML/1998/namespace`. You can use this namespace to define things such as ID attributes (`xml:id`) and languages (`xml:lang`).
- The prefix `xmlns` is bound to `http://www.w3.org/2000/xmlns/`. You can use this namespace to declare XML namespaces.

These namespaces may not be bound by using any other prefix except those defined. Within a document, the prefix `xmlns` must never be declared. The `xml` prefix, on the other hand, may be declared, although it's not necessary. If declared, though, it must be bound to the `http://www.w3.org/XML/1998/namespace` namespace.

Prefixes should also not begin with the characters `xml`. Prefixes that begin with these characters are reserved for future specifications. However, a processor will not treat the use of these as a fatal error, but documents that do use prefixes with these characters may possibly not be valid in the future if a specific prefix ends up being used in any currently undefined specifications.

Namespace Scope

Up until now, you have looked only at namespaces defined in the document element. You can declare namespaces by using any element in the document. So what happens when you encounter additional namespaces? Consider the following document:

```
<Order xmlns:cus="http://www.example.com/Customer"
       xmlns:item="http://www.example.com/Item"
       xmlns="http://www.example.com/Order">
     <cus:Customers>
          <Customer xmlns:cus="http://www.example.com/GENERIC_Customer">
               <cus:Name>John Smith</cus:Name>
               <cus:Number>12345</cus:Number>
          </Customer>
          <cus:Count>1</cus:Count>
     </cus:Customers>
     <item:Items>
          <item1:Item xmlns:item1="http://www.example.com/GENERIC_Item">
               <item1:Name>Book</item1:Name>
               <item1:Number>11111</item1:Number>
          </item1:Item>
          <Item xmlns:item="http://www.example.com/GENERIC_Item">
               <item:Name>Software</item:Name>
               <item:Number>22222</item:Number>
          </Item>
     </item:Items>
     <GeneralInfo xmlns="http://www.example.com/General">
          <Name>General Information</Name>
          <Number>33333</Number>
     </GeneralInfo>
</Order>
```

It's time to play the "Which namespace am I in?" game. You may have been curious why I suggested avoiding using default namespaces if possible. This document is not highly complex because it is quite small and has only a few levels, but it takes namespace use to the extreme—almost to the level of abuse. It should help you to not only understand namespace scoping but also to understand why default namespaces can cause a document to become confusing to read.

What namespace is the `item:Name` element in?

At first glance, you might say `http://www.example.com/Item` because that is the namespace defined on the `Order` element using the `item` prefix. This, however, is wrong. The element is actually in the `http://www.example.com/GENERIC_Item` namespace.

To fully understand how the namespace/element associations are made, you should walk through the document tree and examine the elements. Beginning with the document element, three namespaces are defined:

- `cus` is associated with `http://www.example.com/Customer`.
- `item` is associated with `http://www.example.com/Item`.
- `http://www.example.com/Order` is a default namespace.

The element `cus:Customers` is in the `http://www.example.com/Customer` namespace. This should be obvious, as you have encountered no other namespace definitions. Descending into the content, you encounter the `Customer` element. This element belongs to the `http://www.example.com/Order` namespace. Because it has no prefix and is not defining a default namespace, it inherits the current in-scope default namespace. The element does, however, define a new namespace, `http://www.example.com/GENERIC_Customer`, and it associates the prefix `cus` with it. This prefix used to be associated with `http://www.example.com/Customer`, but for any elements or attributes using this prefix within the contents of the `Customer` element, it now refers to `http://www.example.com/GENERIC_Customer`. This means `cus:Name` and `cus:Number`, which are children of `Customer`, are both in the `http://www.example.com/GENERIC_Customer` namespace.

As you exit from the `Customer` element, the `http://www.example.com/GENERIC_Customer` namespace associated with the `cus` prefix goes out of scope. These were defined on the `Customer` element, which is now closed, so the definition ceases to exist. However, `cus` is now in scope from its definition on the `Order` element. When you encounter the next element, `cus:Count`, it belongs to the `http://www.example.com/Customer` namespace because of the scoping rules. Moving back up the tree, you can safely ignore the `cus:Customers` closing element. Because the element did not define any namespaces, it does not alter anything.

The `item:Items` element is the next element encountered. No changes exist in namespace, so it is bound to the `http://www.example.com/Item` namespace as defined on the `Order` element. Its child element, `item1:Item`, defines the `http://www.example.com/GENERIC_Item` namespace with the `item1` prefix. As this element is also prefixed with `item1`, it ends up in the `http://www.example.com/Item/1` namespace, which it is defining. Both of its children, `item1:Name` and `item1:Number`, will belong to the same `http://www.example.com/GENERIC_Item` namespace defined on their parent.

Entering the second `Item` element, the namespace `http://www.example.com/GENERIC_Item` is once again defined but associated with the `item` prefix. This changes the scope of the prefix so that all the elements contained within `Item` and using the prefix `item` will now be bound to `http://www.example.com/GENERIC_Item` rather than to the one defined on the `Order` element.

The Item element itself has no prefix so is bound to the default namespace, which currently is http://www.example.com/Order. With the newly defined item prefix, both the children elements, item:Name and item:Number, belong to http://www.example.com/GENERIC_Item. Upon leaving the last Item element, the item prefix loses scope, but since it was defined before in an ancestor element (Order), item again refers to the http://www.example.com/Item namespace.

The next element hit is the GeneralInfo element. This demonstrates how it might be confusing to use default namespaces. This element resides in the default namespace. It, however, is also defining a default namespace. The question now arises—to which default namespace does it belong?

Remember the section "Default Namespaces"? Elements defining a default namespace, and not bound to any namespace, will be bound to the default namespace they're defining. To answer the original question then, GeneralInfo is bound to http://www.example.com/General. This also means all elements contained within GeneralInfo will now use http://www.example.com/General as the default namespace. So with that information, there is no way to trick you by asking you what the namespace for the child Name and Number elements are. Of course, they are bound to http://www.example.com/General. When a parser encounters the GeneralInfo closing tag, the default namespace defined on that element falls out of scope, and http://www.example.com/Order comes back into scope as the default namespace of the document.

It's a good thing this was a simple document. Just imagine how hard it would have been to explain a large and complex document. Here are a few tips for writing XML documents:

- If you don't need namespaces, don't use them.
- If you have the choice, use QNames rather than default namespaces.
- Attributes are not bound to default namespaces.
- DTDs and namespaces are not all that compatible and can lead to invalid documents.

Namespaces and Attribute Uniqueness

Back in the "Attributes" section, you learned attributes must be unique for an element. Namespaces add a little twist to this. Attributes names must still be unique, where the name consists of the prefix and local name for a namespaced attribute, but they must also not have the same local name and prefixes that are bound to the same namespace.

In the following example, although the attribute names, a1:z and a2:z, are unique, they are both bound to the same namespace, http://www.example.com/a, which means this is an invalid document:

```
<x xmlns:a1="http://www.example.com/a" xmlns:a2="http://www.example.com/a">
     <y a1:z="1" a2:z="2" />
</x>
```

The following attributes are perfectly legal. The attribute a1:z is bound to http://www.example.com/a1, and a2:z is bound to http://www.example.com/a2.

```
<x xmlns:a1="http://www.example.com/a1" xmlns:a2="http://www.example.com/a2">
     <y a1:z="1" a2:z="2" />
</x>
```

The following example may throw you a bit. Default namespaces do not apply to attributes, so these attributes are unique. Their names are unique because the qualified names are used for comparison, and no duplicate namespace exists. Attribute `a:z` is bound to `http://www.example.com/a`, and attribute `a` is not in any namespace.

```
<x xmlns:a="http://www.example.com/a" xmlns="http://www.example.com/a">
     <y a:z="1" z="2" />
</x>
```

■ **Note** The remainder of the examples in this chapter that use DTDs are well-formed documents but are not valid. If loading them into a parser, make sure you disable validation; otherwise, validation errors will occur. For more information, see Chapter 3.

Using IDs, IDREF/IDREFS, and xml:id

When dealing with documents, it is often useful to be able to uniquely identify elements and be able to easily locate them. Attribute IDs serve this same purpose. When applied to an element, which can have at most a single ID (though this is not the case when using `xml:id`), the value of the attribute on the element serves as the unique identifier for the element. An `IDREF`, on the other hand, allows elements to reference these unique elements.

At first glance, you may be wondering what purpose the `ID` and `IDREF` instances actually serve. Of course, they uniquely identify an element, but what advantage does that offer to you? Before answering that question, I'll cover how you construct them. You can create an attribute ID in two ways. The first is through an attribute declaration (`ATTLIST`) in a DTD. (Chapter 3 covers DTDs in depth; in this chapter, I'll explain `ATTLIST` and its makeup in regard to IDs.) On February 8, 2004, the W3C released the `xml:id` specification as a candidate recommendation. This provides a mechanism to define IDs without requiring a DTD. Since this is relatively new, I will begin with the `ATTLIST` method and then return to `xml:id`.

Defining IDs Using a DTD

Earlier, when discussing the prolog of the document, I touched upon the document type declaration and where it is defined. Similar to Listing 2-4, you can use an internal subset to declare the attribute. Defining attributes takes the following form:

```
<!ATTLIST element_name attribute_name attribute_type attribute_default >
```

In this case, `attribute_type` is the ID. Attribute types, as well as the entire `ATTLIST` definition, are fully explained in Chapter 3, so for now, just take this at face value. You also, for now, will use `#REQUIRED` for `attribute_default`. This just means every element with the name `element_name` is required to have the `ID` attribute named `attribute_name` defined.

Consider the XML document in Listing 2-16, which could serve as a course list for a school.

Listing 2-16. *Course Listing*

```
<Courses>
     <Course id="1">
          <Title>Spanish I</Title>
          <Description>Introduction to Spanish</Description>
     </Course>
     <Course id="2">
          <Title>French I</Title>
          <Description>Introduction to French</Description>
     </Course>
     <Course id="3">
          <Title>French II</Title>
          <Description>Intermediate French</Description>
     </Course>
</Courses>
```

Does this document contain IDs used to uniquely identify elements and for ID lookups?

The answer is no. However, it may appear to do so; since the attribute name is `id` and the values of the attributes are unique, the attributes within the document are just plain, everyday attributes. This is a problem many people frequently encounter, and I have fielded many bug reports claiming that IDs are not working properly in a document. The fact is, just creating an attribute with the name `ID` does not make it an ID. IDs can actually be named anything you like, assuming it is a legal XML name. The document must somehow be told that the attribute is of type `ID`. There is also a caveat about the allowed values for attribute IDs. The values must follow the rules for legal XML names. So within the previous example, the value `1` is invalid because names cannot begin with a number.

■ **Caution** An attribute with the name `ID` is not automatically an ID. You must make the document aware that an attribute is of type `ID`. Once identified, the values of the attribute IDs must conform to the rules defined by legal XML names and so may not begin with a number.

Listing 2-17 shows how to rewrite the document so it can use IDs.

Listing 2-17. *New Course Listing*

```
<!DOCTYPE Courses [
     <!ATTLIST Course cid ID #REQUIRED>
]>
<Courses>
     <Course cid="c1">
          <Title>Spanish I</Title>
          <Description>Introduction to Spanish</Description>
     </Course>
```

```
    <Course cid="c2">
        <Title>French I</Title>
        <Description>Introduction to French</Description>
    </Course>
    <Course cid="c3">
        <Title>French II</Title>
        <Description>Intermediate French</Description>
    </Course>
</Courses>
```

Comparing the documents from Listing 2-16 and Listing 2-17, you will notice that I added a document type declaration and I named the attributes `cid`. I changed the name to illustrate that you can use any valid names for IDs and not just `id`. I added the `ATTLIST` declaration to define the attributes named `cid` when applied to elements named `Course` of type `ID` and to define that the attribute is required for all `Course` elements. You may also notice that the values for the attributes have changed. With respect to the rules surrounding the attribute value, I prefixed the numeric values with the letter *c* so they conform to the rules for legal XML names.

After the document in Listing 2-17 has been parsed, you will end up with two `Course` elements that are uniquely identified by the value of the `cid` attribute. Now I can answer the original question of what purpose they serve. The answer really depends upon what you are doing. For instance, if you were to load the document under the DOM, using the DOM Document object, you could retrieve specific elements by calling the `getElementById()` method. Passing in the unique value as the parameter to the method, such as `c2`, the `Course` element that contains information on French I would be returned. Distinct elements could also be returned using XPath queries, such as those used in XSL. IDs can also be referenced within a document, which brings us to `IDREF`.

IDREF

An `IDREF` is a method that allows an element to reference another element. It is basically a pointer from one element to another. Taking the course list in Listing 2-17, how could you expand it to add course prerequisite information? One way to do this would be to duplicate the course information for the prerequisites, as shown in Listing 2-18.

Listing 2-18. *Course Listing with Prerequisites*

```
<!DOCTYPE Courses [
    <!ATTLIST Course cid ID #REQUIRED>
]>
<Courses>
    <Course cid="c2">
        <Title>French I</Title>
        <Description>Introduction to French</Description>
    </Course>
    <Course cid="c3">
        <Title>French II</Title>
        <Description>Intermediate French</Description>
        <pre-requisite>
```

```
                <Pcourse>
                    <Title>French I</Title>
                    <Description>Introduction to French</Description>
                </Pcourse>
            </pre-requisite>
        </Course>
</Courses>
```

This is not an efficient way of handling data. The element name `Course` could not be used for the prerequisite. Course elements require the `ID` attribute `cid`, but for this document, the prerequisites should not be IDs. This could be handled by changing the `attribute_type` in the `ATTLIST`, covered in Chapter 3, but this still requires duplicating the content for the French I course. No correlation within the document exists that says the `Course` element containing French I in the prerequisites is the same as the `Course` element identified by `c2`.

Modifying the document in Listing 2-18, you can add an `IDREF`, as shown in Listing 2-19. For now, the document continues to use `Pcourse` for the element name.

Listing 2-19. *Course Listing with Prerequisites Using* `IDREF`

```
<!DOCTYPE Courses [
    <!ATTLIST Course cid ID #REQUIRED>
    <!ATTLIST Pcourse cref IDREF #REQUIRED>
]>
<Courses>
    <Course cid="c2">
        <Title>French I</Title>
        <Description>Introduction to French</Description>
    </Course>
    <Course cid="c3">
        <Title>French II</Title>
        <Description>Intermediate French</Description>
        <pre-requisite>
            <Pcourse cref="c2" />
        </pre-requisite>
    </Course>
</Courses>
```

`Pcourse` no longer contains all the additional baggage and redundant data. The `IDREF`, `cref`, now refers to the `Course` element identified by `c2`. The document no longer contains redundant data, making it more compact as well as easier to read. In addition, you can reuse the content. Imagine how long the document would be if you created an entire school course list, along with all prerequisites, without using IDs and `IDREF`.

IDREFS

Sometimes an element will need to reference more than one ID of the same element type. For example, in Listing 2-19, it would be much easier if the `pre-requisite` element could reference the courses directly, rather than adding child elements for the courses. Multiple attributes of

the same name are not allowed for an element, so you must use IDREFS to perform this feat, as shown in Listing 2-20.

Listing 2-20. *Course Listing with Prerequisites Using* IDREFS<!DOCTYPE Courses [

```
     <!ATTLIST Course cid ID #REQUIRED>
     <!ATTLIST pre-requisite cref IDREFS #REQUIRED>
]>
<Courses>
     <Course cid="c1">
          <Title>Basic Languages</Title>
          <Description>Introduction to Languages</Description>
     </Course>
     <Course cid="c2">
          <Title>French I</Title>
          <Description>Introduction to French</Description>
     </Course>
     <Course cid="c3">
          <Title>French II</Title>
          <Description>Intermediate French</Description>
          <pre-requisite cref="c1 c2" />
     </Course>
</Courses>
```

You will notice that the element pre-requisite now contains a single attribute, cref, with the value c1 c2. The value of the IDREFS attribute is a whitespace-delimited list of IDREF. This means cref is a pointer to *both* the Course element identified by c1 and the Course element identified by c2.

Using xml:id

In 2004, the W3C released the xml:id specification as a recommendation. Using xml:id within a document allows you to create IDs without requiring a DTD. This is a much easier method than creating attribute declarations, though the two have a few differences:

- The values for xml:id *must* conform to legal namespace names. This is almost identical to regular IDs, except a colon is not a valid character for the value.
- When defined in a DTD, though not a requirement to do so, xml:id *must* be defined as an ID. The attribute type for xml:id cannot be modified to another type.

Re-creating the course list from Listing 2-17, using xml:id rather than declaring attributes of type ID, the document would look as follows:

```
<Courses>
     <Course xml:id="c1">
          <Title>Spanish I</Title>
          <Description>Introduction to Spanish</Description>
     </Course>
```

```
     <Course xml:id="c2">
          <Title>French I</Title>
          <Description>Introduction to French</Description>
     </Course>
     <Course xml:id="c3">
          <Title>French II</Title>
          <Description>Intermediate French</Description>
     </Course>
</Courses>
```

To use an `IDREF`, however, the `IDREF` still must be declared in the DTD. So, re-creating the document in Listing 2-18 using `xml:id` and `IDREF`, the document would take this form:

```
<!DOCTYPE Courses [
     <!ATTLIST Pcourse cref IDREF #REQUIRED>
]>
<Courses>
     <Course xml:id="c2">
          <Title>French I</Title>
          <Description>Introduction to French</Description>
     </Course>
     <Course xml:id="c3">
          <Title>French II</Title>
          <Description>Intermediate French</Description>
          <pre-requisite>
               <Pcourse cref="c2" />
          </pre-requisite>
     </Course>
</Courses>
```

You don't need to do anything else to handle IDs using `xml:id`. As I said before, it is simple to use and is great when you don't want to deal with DTDs. One less thing to complicate the document is always better!

Using xml:space and xml:lang

Two special attributes that are part of the XML specification can provide additional information to a document about how certain things should be processed: `xml:space` and `xml:lang`. These are not like PIs, which are application specific. These attributes, being part of the XML specification, are meant to be handled by any application. When using these attributes within a document to be validated, you must define attribute declarations for these attributes within the DTD; otherwise, validation errors may occur.

xml:space

This attribute specifies to an application how it should handle whitespace. The valid values are `preserve` and `default`. When set to `default`, the application handles whitespace as it normally does. A value of `preserve` instructs the application that it must preserve all white-space within the context of the element on which the attribute is set. For example:

```
<Description xml:space="preserve">
<a>This</a>
     <b>is</b>
          <c>the</c>
               <d>description</d>
</Description>
```

The value of `preserve` should instruct the application to preserve the whitespace within the description content. If this were set to `default`, the application may or may not preserve whitespace. It would depend upon its default behavior.

xml:lang

The `xml:lang` attribute can specify the language used for the content within an element. The values can come from the ISO standard 639, denoted by the IANA prefix `i-`, or from private sources, denoted by the prefix `x-`. For example:

```
<docu xml:lang="en">
    <p xml:lang="fr">Bonjour monde en français </p>
    <p xml:lang="de">Hallo Welt auf Deutsch<p>
    <p>Hello World in English</p>
</docu>
```

The document illustrates "Hello World" in French (`xml:lang="fr"`), German (`xml:lang="de"`), and English. The `p` tag for English has no `xml:lang` attribute because it is in the scope of the `docu` element, which is set to `xml:lang="en"`. Therefore, unless overridden, the default content of the `docu` element is in English.

Understanding XML Base

Unlike `xml:space` and `xml:lang`, XML Base is not part of the XML specification. It has its own specification from the W3C. The `xml:base` attribute specifies a base URI on an element, which is used to resolve relative URIs used within the scope of the element. The use of `xml:base` may also be stacked. By this I mean that within the scope of an element defining an `xml:base`, an element may define a relative URI as its `xml:base`. This would effectively set the base URI within the context of this subelement as the path of this new base, relative to the ancestor base URI.

XML Base is primarily used for XLink to describe linking between resources. You may also see it used in other contexts, such as with XInclude and XSLT. The following is a document that uses XInclude to illustrate how `xml:base` can define base URIs for the XInclude documents:

```
<example xmlns:xi="http://www.w3.org/2001/XInclude">
     <para xml:base="http://www.example.com/">
          <xi:include href="example.xml" />
          <p2 xml:base="examples/">
               <xi:include href="example1.xml" />
          </p2>
          <p3>
               <xi:include href="examples/example1.xml" />
          </p3>
     </para>
</example>
```

Within the `para` element, the base URI is set to `http://www.example.com/`. Everything within the scope of this element will now use this URI as the base for any relative URI. As you descend into the child elements, the first `xi:include` points to `example.xml`. This will resolve to `http://www.example.com/example.xml` when included in the document.

Moving to the `p2` element, `xml:base` is set to `examples/`. This is a relative URI, so for all practicality, it inherits the base of the encapsulating element's URI (`http://www.example.com/`) and sets the base relative to this. The base is now `http://www.example.com/examples/` for the `p2` element and everything within its scope. When the `xi:xinclude` element is reached within this element, the file `example1.xml` will resolve to `http://www.example.com/examples/example1.xml` when included.

Continuing to navigate the document, you reach the end of `p2`. The base that was set falls out of scope, which means the base set by the `para` element, `http://www.example.com/`, becomes the active base again. Upon reaching the `xi:include` within the `p3` element, the file `examples/example1.xml`, being relative, uses the base URI from `para` and resolves to `http://www.example.com/examples/example1.xml` when included. This is the same file that `p2` had included, just using relative pathing a little differently based upon the scope of `xml:base` within the document.

Conclusion

This chapter covered the basic structure, syntax, and a few other areas of XML that will help you understand documents, regardless of their complexity. Although a few more complex aspects of XML exist, you should be well on your way to creating well-formed XML documents with the basics presented here. The next chapter will introduce you to validating with DTDs, XML Schemas, and RELAX NG. What you have learned in this chapter will be invaluable to you throughout the rest of this book.