Apress™
Books for Professionals by Professionals

Chapter Fourteen: "PHP and XML"

A Programmer's Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

CHAPTER 14

PHP and XML

It can hardly be argued that the Web has not vastly changed the landscape on which we share information. The sheer vastness of this electronic network has made the establishment of certain standards not only a convenience, but a requirement if organizations are ever going to exploit the Web to its fullest capability. XML (eXtensible Markup Language) is one such standard, providing a means for the seamless interchange of data between organizations and their applications. The implications of this are many, resulting in the facilitation of media-independent publishing, electronic commerce, customized data retrieval, and many other data-oriented services.

In the first part of this chapter, I provide a general introduction to XML, highlighting the general syntactical elements that comprise the language. The second half of this chapter is dedicated to PHP's XML-parsing capabilities, elaborating on its predefined XML functionality and the language's general XML-parsing process. This material is geared toward providing you with a better understanding of both why XML is so useful and how you can begin coming to terms with how PHP can be used to develop useful and interesting XML-based applications.

Before delving directly into the issue of XML, many newcomers to this subject may find it useful to learn more about the history behind the concepts that ultimately contributed to the development of the XML standard.

A Brief Introduction to Markup

As its name so implies, HTML (HyperText Markup Language) is what is known as a *markup language*. The term *markup* is defined as the general description for the document annotation that, instead of being displayed to whatever media the document is destined for, is used for describing *how* parts of that document should be formatted. For example, you may want a particular word to be **boldfaced** and another *italicized*. You may wish to use a particular font for one paragraph and a larger font size for a header. As I type this paragraph, my word processor is using its own form of *markup* in order to properly present the formatting as I specify it to be. Therefore, the word processor is using its own particular *formatting markup language* implementation. In short, the markup language used by my word processor is a means for specifying the visual format of the text in my document.

There are many types of markup languages in the world today. For example, communication applications use a form of markup to specify the meaning of each group of 1's and 0's sent over the Internet. Humans use a sort of markup language when underlining or crossing out words in a textbook. Regardless of its format, a markup language accomplishes two important tasks:

- **It defines what is considered to be valid markup syntax.** In the case of the HTML specification, *<b>text</b>* would be a valid markup statement, but *<xR5t>text</x4rt>* would be invalid, due to mismatching opening and closing tags.

- **It defines what is meant by a particular valid markup syntax.** Surely you know that *<b>text</b>* is an HTML command to format in boldface the word *text*. That is an example of the markup defining what is to result when a particular markup *document component* is declared.

HTML is a particularly popular markup language, as is obvious when watching the explosive growth of the Web over the past few years. But how was this language derived? Who thought to use tags such as <b> and </b> to specify meaning in a document? The answer to this lies in HTML's forefather, SGML (Standard Generalized Markup Language).

The Standard Generalized Markup Language (SGML)

SGML is an internationally recognized standard for exchanging electronic information between varied hardware and software implementations. Judging from its name, you would think that SGML is some sort of language. This is perhaps a bit misleading, since SGML is actually defined as a formalized set of rules from which languages can be created. Two particularly popular languages derived from SGML are HTML and XML. As you already know, HTML is a platform- and hardware-independent language used to format and display text. The same is true of XML.

SGML was born out of the necessity to share data between different applications and operating systems. As far back as the 1960s, this was already fast becoming a problem for computer users. Realizing the constraints of the many nonstandard markup languages, three IBM researchers, Charles Goldfarb, Ed Mosher, and Ray Lorie, began unearthing three general concepts that would make it possible to begin sharing documents across operating systems and applications:

- **The document-processing programs must all be able to communicate using a common formatting language.** This makes sense, since we know from our own experiences that communication among individuals speaking different languages is difficult. However, if we are all provided with the same set of syntax and semantics, communication becomes much easier.

- **The formatting language should be specific to its purpose.** The ability to custom-build a language based on a particular set of predefined rules frees the developer from having to depend on a third-party implementation of what is assumed that the end user requires.

- **The document format must closely follow a set of specific rules.** These rules relate to such things as the number and label of the language constructs used in the document. A standard document format ensures that all users know exactly what the structural outline of that document contains. This last pillar of document sharing is particularly important because it does *not* specify how the document is displayed. Rather, it specifies how the document is structurally formatted. The set of rules used to create this document format is better known as a *document type definition*, or DTD.

These three rules form the basis for SGML's predecessor, Generalized Markup Language, or GML. Research and development of GML continued over the next decade or so, until SGML was born out of an agreement made by an international group of developers.

As the need for a common ground for information exchange became increasingly prevalent in the 1980s, SGML soon became the industry standard (1986 was the year that SGML became an ISO standard) for making it happen. In fact, the standard is still going strong today, with agencies in charge of maintaining enormous amounts of information relying on SGML as a dependable and convenient means for data storage. To put it in perspective, the U.S. Patent and Trademark Office (http://www.uspto.gov), U.S. Internal Revenue Service (http://www.irs.gov), and Library of Congress (http://lcweb.loc.gov) are all prominent users of SGML in their mission-critical applications. Just imagine the amount of documentation that each of these agencies handles each year!

TIP *Arguably the best resource on the Internet for learning more about SGML, XML, and various other markup languages is the Robin Cover/OASIS XML Cover Pages at http://www.oasis-open.org/cover/.*

The idea of passing hypertext documents via a Web browser, as was envisioned by Tim Berners-Lee, did not require many of the features offered by the robust SGML implementation. This resulted in the creation of a well-known markup language called HTML.

The Advent of HTML

Interestingly, the concept of the World Wide Web fit only too perfectly in the idea of using a generalized markup language to facilitate information exchange in an environment harboring a multitude of different hardware, operating system, and software implementations. And in fact, Berners-Lee must have had this matter in mind, as he modeled the first version of HTML after the SGML standard. HTML shares several of SGML's characteristics, including a simple generalized tag set and the angled bracket convention. These simple documents could be effectively read on any computer system, offering a means for viewing text documents. And the rest is history.

However, HTML suffers from the major drawback that it does not offer developers the capability of creating their own document types. This resulted in the onset of the "browser wars," where browser developers begin building their own enhancements to the HTML language. These HTML add-ons severely detracted from the idea of working with a unique HTML standard, not to mention wreaking havoc for developers wishing to create cross-browser Web sites. Furthermore, years of a lax definition standard resulted in developers greatly stretching the boundaries of the original intent of the language. I would not be surprised if the vast majority of Web pages on the Internet today failed to comply with the current HTML specification.

The W3C's (`http://www.w3.org`) reaction to this rapidly worsening situation began with a concerted attempt to steer HTML development back toward the right path: that is, a return to the underlying foundations of SGML. The result of their concentrated efforts? XML.

Irrefutable Evidence of Evolution: XML

XML is essentially the culmination of the efforts of the W3C to offer an Internet-based standard that is in conformance with the three major principles of SGML, first introduced in the previous section, "The Standard Generalized Markup Language (SGML)." Like SGML, XML is not in itself a language; it too is composed of a standard set of guidelines from which other languages can be derived. More specifically, XML is the product of the conglomeration of three separate specifications:

- **XML (Extensible Markup Language)**: This specification defines the core XML syntax.

- **XSL (Extensible Style Language)**: XSL is a specification geared toward separating page style from page content through the practice of applying separate style sheets to documents to satisfy specific formatting requirements.

- **XLL (Extensible Linking Language)**: XLL specifies how links between resources are represented.

XML not only makes it possible for developers to create their own custom languages for Internet application production; it also allows for the validation of these documents for conformance to the XML specification. Furthermore, XML truly promotes the idea of implementation-independent data, since the XSL can be used to specify exactly how the document will be displayed. For example, assume that you have reformatted your Web site to be stored as XML source. You could use a "wireless" style sheet to format the XML source for use on a PDA, such as a Palm Pilot, and another “”personal computer” style sheet to format it for display on a regular computer monitor. Remember, it's the same XML source, just formatted differently to suit the user's device.

NOTE *The Wireless Markup Language (WML) is an example of a popular language derived from XML.*

An Introduction to XML Syntax

Those of you already familiar with SGML or HTML will find the structure of an XML document to be nothing new. Consider Listing 14-1, which illustrates a simple XML document.

Listing 14-1: A simple XML document

```
<?xml version="1.0"?>
<!DOCTYPE cookbook SYSTEM "cookbook.dtd">
<cookbook>
<recipe category="italian">
<title>Spaghetti alla Carbonara</title>
<description>This traditional Italian dish is sure to please even the most
discriminating critic.</description>
<ingredients>
<ingredient>2 large eggs</ingredient>
<ingredient>4 strips of bacon</ingredient>
<ingredient>1 clove garlic</ingredient>
<ingredient>12 ounces spaghetti</ingredient>
<ingredient>3 tablespoons olive oil</ingredient>
</ingredients>
<process>
<step>Combine oil and bacon in large skillet over medium heat. Cook until bacon is
brown and crisp.</step>
<step>Whisk eggs in bowl. Set aside.</step>
```

```
<step>Cook pasta in large pot of boiling water to taste, stirring occasionally.
Add salt as necessary.</step>
<step>Drain pasta and return to pot, adding whisked eggs. Stir over medium-low
heat for 2-3 minutes.</step>
<step>Mix in bacon. Season with salt and pepper to taste.</step>
</process>
</recipe>
</cookbook>
```

There you have it! Your first XML document. Now turn your attention toward the following components of just such a document, elaborating on parts of Listing 14-1 to illustrate their usage:

- XML prolog
- Tag elements
- Attributes
- Entity references
- Processing instructions
- Comments

XML Prolog

All XML documents must begin with a document prolog. This line basically says that XML will be used to build the document and which version of XML will be used to do so. Since the current XML version is 1.0, all of your XML documents should begin with:

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

The next line of Listing 14-1 points to an external DTD. Don't worry too much about this right now. I introduce DTDs in detail in the upcoming section "The Document Type Definition (DTD)."

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

The rest of Listing 14-1 contains elements very similar to those of an HTML document. The first element, cookbook, is what is known as the root element, since its tag set encloses all of the other tags in the document. Of course, you can name your root element whatever you like. The important thing to keep in mind is that its tag set encloses all other elements.

There are other instructions that could be placed in the prolog. For example, you could extend the first above-described declaration by specifying that the document is complete by itself:

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

Setting standalone to "yes" tells the parser that no other files should be imported into this document, such as a DTD.

Although this extension and others are certainly useful, I'll keep document syntax to a minimum in order to better illustrate the central topic of this chapter: how PHP and XML work together.

Elements

The rest of the document consists largely of varied elements and corresponding data. Elements are easily identified, as they are enclosed within angle brackets like those in HTML markup. An element may be empty, consisting of only one tag set, or it may contain information, in which case it must have an opening and closing tag. If it is not empty, then the tag names describe the nature of the informational data (also known as CDATA) enclosed in the tags. As you can see from Listing 14-1, these tags are very similar to those in an HTML document. However, there are a few important distinctions to keep in mind:

- All XML elements must consist of both an opening and closing tag.

- Those elements that are not empty consist of both opening and closing tags. Those tags that would not logically have a closing tag can use an alternative form of syntax <element />. At first, you may wonder what tag would not have a complement. Keep in mind that certain HTML formatting tags like <br>, <hr>, and <img> don't have closing tags. Tags of the same format can be created in XML documents.

- **XML elements must be properly nested**. Listing 14-1 illustrates an XML document that is properly nested; that is, no element tags appear where they shouldn't. For example, you couldn't do the following:

  ```
  <title>Spaghetti alla Carbonara

  <ingredients></title>
  ```

Other than not making sense, it just doesn't make for good form. Subsequent parsing of this XML document would fail.

- **XML elements are case-sensitive** Those of you used to cranking out HTML at 3 a.m. won't like this rule too much. In XML, the tag <tag> is different from <Tag> is different from <TAG>. Get used to it, or this will soon drive you crazy.

Attributes

Just as HTML tags can be assigned attributes, so can XML tags. In short, *attributes* provide further information about the content that could later be used for formatting or processing the XML. These attributes are assigned in name-value pairs, and unlike in HTML, XML attributes *must* be properly enclosed in either single or double quotation marks, or subsequent parsing will fail. Listing 14-1 contains one such element attribute:

```
<recipe category="italian">
```

This attribute basically says that the category of this particular recipe is italian. This could facilitate subsequent grouping and organizational operations.

Entity References

Entities are a way to facilitate document maintenance by referencing some content through the use of some keyword. This keyword could point to something as simple as an abbreviation expansion or as complicated as an entirely new piece of XML content. The convenience in entities lies in the fact that they can be used repeatedly throughout an XML document. When this document is later parsed, all references to that entity will be replaced with the content referred to in the entity declaration. The entity declaration is placed in the DTD referred to by the XML document.

You can refer to an entity in your XML document by calling its name, preceded by an ampersand (&), and followed by a semicolon (;). For example, assume that you had declared an entity that pointed to copyright information. Throughout the XML document, you could then refer to this entity by using the following syntax:

```
&Copyright;
```

Using this in an applicable manner, a line of the XML document might read:

```
<footer>
…various other footer information…
&Copyright;
</footer>
```

Like variables or templates, entities are useful when a certain piece of information may change in the future or continued explicit referencing of that information is too tedious a process to repeat. I'll delve further into the details of referencing and declaring entities in the upcoming section "The Document Type Definition (DTD)."

Processing Instructions

Processing instructions, commonly referred to as *PIs*, are external commands that are used by the application that is working with the XML document. The general syntax for a PI is:

```
<?PITarget instructions?>
```

`PITarget` specifies which application should make use of the ensuing `instructions`. For example, if you wanted PHP to execute a few commands in an XML document, you could make use of a PI:

```
<?php print "Today's date is: ".date("m-d-Y");?>
```

Processing instructions are useful because they make it possible for several applications to work with the same document in unison.

Comments

Comments are always a useful feature of any language. XML comment syntax is exactly the same as that of HTML comment syntax:

```
<!— Descriptive comments go here —>
```

Okay, so you've seen your first XML document. However, there is another very important aspect of creating valid XML documents: the document type definition, or DTD.

The Document Type Definition (DTD)

A *DTD* is a set of syntax rules that form the basis for validation of an XML document. It explicitly details an XML's document structure, elements, and element attributes, in addition to various other pieces of information relevant to any XML document derived from that DTD.

Keep in mind that it is not a requirement that an XML document has an accompanying DTD. If a DTD does exist, then the XML system can use this DTD as a reference for how to interpret the XML document. If a DTD is not present, it is assumed that the XML system will be able to apply its own rules to the document. However, chances are that you want to include a DTD with your XML document to verify its structure and interpretation.

A DTD may be placed directly in the XML document itself, referenced via a URL or via some combination of both methods. If you wanted to place the DTD directly in the XML document, you would do this by defining the DTD directly after the prolog as follows:

```
<!DOCTYPE root_element_name [
…various declarations…
] >
```

The reference to `root_element_name` will correspond to the name of the root element surrounding your XML document. The section specified by "`various declarations`" is where the element, attribute, and various other declarations are defined.

Chances are you will want to place your DTD in a separate file to facilitate modularity. Therefore, let's begin by showing how a DTD can be referenced from within an XML document. This is accomplished with a simple command:

```
<!DOCTYPE root_element_name SYSTEM "some_dtd.dtd">
```

As was the case with the internal DTD declaration, `root_element_name` refers to the name of the root element surrounding your XML document. The keyword SYSTEM refers to the fact that `some_dtd.dtd` is located on the local server. You could also point to `some_dtd.dtd` by referring to its absolute URL. Finally, the URL referenced in quotations points to the external DTD. This DTD could reside either locally or on some other server.

So how would you create a DTD for Listing 14-1? First of all, you want to call the DTD from within the XML document. As discussed in the previous section, the DTD is referenced with the following command:

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

Looking back to Listing 14-1, you see that cookbook is the `root_element_name`. The name of the DTD being referenced is cookbook.dtd. The DTD itself is shown in Listing 14-2. A line-by-line description of the listing ensues.

Listing 14-2: DTD for Listing 14-1, entitled "cookbook.dtd"

```
<?xml version="1.0"?>
<!DOCTYPE cookbook [
<!ELEMENT cookbook (recipe+)>
<!ELEMENT recipe (title, description, ingredients, process)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ingredients (ingredient+)>
<!ELEMENT ingredient (#PCDATA)>
<!ELEMENT process (step+)>
<!ELEMENT step (#PCDATA)>
   <!ATTLIST recipe category CDATA  #REQUIRED>
] >
```

So what does this rather strange-looking document mean? Although seemingly cryptic at first, it is actually rather simple. Let's go over Listing 14-2 line by line:

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

The first line is essentially the XML prolog. You have already been introduced to this.

```
<!DOCTYPE cookbook [
```

The second line states that a DTD is beginning, and the DTD title is cookbook.

```
<!ELEMENT cookbook (recipe+)>
```

The third line refers to an actual tag element in the XML document, in this case the root element, which is cookbook. Immediately following is the word recipe enclosed in parentheses. This means that enclosed in the cookbook tags will be a *child* tag element named recipe. The plus sign following recipe means that there will be at least one set of the recipe tags in the *parent* cookbook tags.

```
<!ELEMENT recipe (title, description, ingredients, process)>
```

The fourth line defines the recipe tag. It states that in the recipe tag, four distinct child tags will be found: title, description, ingredients, and process. Since no occurrence indicators (more about occurrence indicators in the following section, "DTD Components") follow any of the tag declarations, it is assumed that one set of each will appear in the recipe tag.

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

Here we happen on the first tag definition that does not contain any nested tags. Instead it is said to hold #PCDATA. The keyword #PCDATA stands for character data, that is, any data that is not considered to be markup oriented.

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

The element definition of description, like title, states that the description tags will not hold anything else except character data.

```
<!ELEMENT ingredients (ingredient+)>
```

The definition of the ingredients element states that it will contain one or more tags named ingredient. Check out Listing 14-1, and you will realize how logical this is.

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

Since the tag element ingredient refers to a single ingredient in the list, it only makes sense that this element will contain character data.

```
<!ELEMENT process (step+)>
```

The element process is expected to contain one or more instances of the element step.

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

The element step, like ingredient, is a component of a larger list. Therefore, it is expected to contain character data.

```
<!ATTLIST recipe category CDATA  #REQUIRED>
```

Notice that the recipe element in Listing 14-1 contains an attribute. This attribute, category, refers to a general category in which the recipe would fall, in this case Italian. Note that both the element name and the attribute name are specified in this ATTLIST definition. Furthermore, because of the fact that for referential purposes it would be useful to categorize every single recipe, we specify that this attribute is #REQUIRED.

```
] >
```

This final line simply closes the DTD definition. You must always properly enclose the definition, or an error will occur.

Let's finish this section with a synopsis of the major components of a typical DTD:

- Element type declarations
- Attribute declarations
- ID, IDREF, and IDREFS
- Entity declarations

You were introduced to several of these components in the preceding review of Listing 14-2. Now I'll cover each component in further detail.

Element Declarations

All elements used in an XML document must be properly defined if a DTD accompanies the document. You've already seen two commonly used element definition variations: defining an element to contain other elements, and defining an element to contain character data. To recap, the following definition of the tag element description specifies that it will contain only character data:

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

The following definition of the element process specifies that it will contain exactly one occurrence of the element named step:

```
<!ELEMENT process (step)>
```

Of course, it might not make too much sense to just have one step in a process, and chances are you would have more. Therefore you can use the occurrence indicator to specify that there will be *at least* one occurrence of the element step:

```
<!ELEMENT process (step+)>
```

You can specify the frequency of occurrence of elements in several different ways. A listing of available element operators is shown in Table 14-1.

Table 14-1. Element Operators

INDICATOR	**MEANING**
?	Zero or one occurrences
*	Zero or more occurrences
+	One or more occurrences
[none]	Exactly one time
|	Either element
,	The first element must follow the second element.

If you intended on including several different tags in a specific tag element, you delimit each with a comment in the element definition:

```
<!ELEMENT recipe (title, description, ingredients, process)>
```

Since there are no occurrence indicators, each of these tags must appear *only once*.

You can also use Boolean logic to further specify the definition of an element. For example, assume that you were dealing with recipes that always specified pasta accompanied with one or more types of either cheese or meat. You could define the ingredient element as follows:

```
<!ELEMENT ingredient (pasta+, (cheese | meat)+)>
```

Since you always want the pasta tag to appear, you place the plus (+) occurrence indicator after it. Then, either the cheese *or* meat element is expected; therefore you separate them with a vertical bar and proceed the parentheses block with a plus (+), since one or the other is always expected.

There are many other element definition variations. This is only the beginning. However, what has been covered thus far should suffice for you to effectively follow the examples presented throughout the rest of this chapter.

Attribute Declarations

Element attributes describe what kind of value an element may have. Like HTML tag elements, XML elements may have zero, one, or several attributes. The general syntax for an attribute declaration is:

Gilmore_14 12/5/00 10:25 AM Page 369

```
<!ATTLIST element_name
attribute_name1 datatype1 flag1
…
>
```

Where `element_name` is the name of the tag element. The attributes for this tag element then ensue. There are three main components of each attribute, the name, specified by `attribute_name1`; its datatype, specified by `datatype1`; and a flag specifying how that attribute value is handled, specified by `flag1`. The ellipsis (…) signifies that more than one attribute declaration can be placed here.

You've already seen a simple example of an attribute declaration in Listing 14-2:

```
<!ATTLIST recipe category CDATA  #REQUIRED>
```

However, as you can see from the general syntax definition, you can also simultaneously declare multiple attributes. For example, suppose that you wanted to assign the recipe element not only a category attribute, but a difficulty (in preparation) attribute as well. This would be a multiple-attribute declaration. You could declare both of these attributes in the same list:

```
<!ATTLIST recipe category CDATA  #REQUIRED
                              difficulty CDATA  #REQUIRED>
```

You are not required to format the declaration as I've done; However, it improves readability over just letting the declarations run together on a single line. Also, since both attributes are required, you cannot just use the recipe tag with only one or the other; both must be used. For example, this would be wrong:

```
<recipe difficulty="hard">
```

Why? Because the category attribute is not present. However, this would be correct:

```
<recipe category="Italian" difficulty="hard">
```

There are actually three different flags that can be used to indicate how an attribute value is handled. These flags and their descriptions are shown in Table 14-2.

Table 14-2. Attribute Flags

FLAG	DESCRIPTION
#FIXED	Specifies that the attribute can only be assigned one specific value for every element instance in the document.
#IMPLIED	Specifies that a default attribute value can be used if the attribute is not included with the element.
#REQUIRED	Specifies that the attribute is not optional and must always be present with each element instance.

Attribute Types

An element attribute can be declared as one of a number of types. Each type is described in further detail in this chapter.

CDATA Attributes

Many times, you will be interested in just ensuring that the attributes contain general character data. These are known as CDATA attributes. The following example was already shown at the beginning of this section:

```
<!ATTLIST recipe category CDATA  #REQUIRED>
```

ID, IDREF, and IDREFS Attributes

Throughout several chapters of this book I introduced the idea of using identification numbers to uniquely identify data, such as user or product information stored in a database table. The use of unique IDs is also particularly useful in the world of XML, since cross-referencing information across documents is common not only in general information management but also on the World Wide Web (via hyperlinks).

Element IDs are assigned the ID attribute. For example, assume that you want to assign each recipe a unique identification number. The DTD syntax might look like the following:

```
…
<!ELEMENT recipe (title, description, ingredients, process)>
<!ATTLIST recipe recipe-id ID #REQUIRED>
<!ELEMENT recipe-ref EMPTY>
<!ATTLIST recipe-ref go IDREF #REQUIRED>
…
```

You could then declare the recipe element in a document as follows:

```
<recipe recipe-id="ital003">
<title>Spaghetti alla Carbonara</title>
…
```

The identifier ital003 uniquely identifies this recipe. Keep in mind that since recipe-id is of type ID, the same identifier cannot be used in any other recipe recipe-id value, or the document will be invalid. Now suppose that later on you want to reference this recipe somewhere else, for example, in a user's list of favorite recipes. This is where the element cross-reference and the IDREF attribute come into play. IDREF can be assigned an ID value for referring to the element specified by ID, kind of like a hyperlink refers to a page specified by a particular URL. Consider the following XML snippet:

```
<favoriteRecipes>
<recipe-ref go="ital003">
</favoriteRecipes>
```

Once the XML document is parsed, the recipe-ref element would be replaced with a more user-friendly reference pointing to the recipe having that ID, such as the recipe title. Also, it would probably be formatted as a hyperlink to facilitate navigation to that recipe.

Enumerated Attributes

You can also specify a restricted list of potential values for an attribute. This would actually work quite well to improve the above declaration, since you could assume that you would have a specific list of recipe categories and could limit the levels of difficulty to a select few adjectives. Let's refine the previous declaration to read:

```
<!ATTLIST recipe category (Italian | French | Japanese | Chinese)  #REQUIRED
                              difficulty (easy | medium |  hard) #REQUIRED>
```

Notice that when using restricted value sets, you are no longer required to include CDATA. This is because all of the values are already of CDATA format.

Default Enumerated Attributes

It is sometimes useful to declare a default value. Chances are you have probably done this in the past when building forms that have drop-down lists. For example, if the majority of your recipe submissions are from Italians, chances are the majority of the recipes will be of the Italian category. You could set Italian as the default category like this:

```
<!ATTLIST recipe category (Italian | French | Japanese | Chinese) "Italian">
```

In the above declaration, if no other category value has been set, then the category will automatically default to Italian.

Entities and Entity Attributes

Not all of the data in an XML document is necessarily text based. Binary data such as graphics may appear as well. This data can be referred to by using entity attributes. You could specify that a (presumably) graphic named recipePicture will appear within the description element as follows:

```
<!ATTLIST description recipePicture ENTITY #IMPLIED>
```

Similarly, you could simultaneously declare several entities by using the entities attribute in place of the entity attribute. Each ENTITY value is separated by white space.

NMTOKEN and NMTOKENS Attributes

An NMTOKEN, or name token, is a string composed of a restricted range of characters. Therefore, declaring an attribute to be of type NMTOKEN would suggest that the attribute value be in accordance with the restriction posed by NMTOKEN. Typically, an NMTOKEN attribute value consists of only one word:

```
<!ATTLIST recipe category NMTOKEN  #REQUIRED>
```

Similarly, you could simultaneously declare several entities by using the NMTOKENS attribute in place of the NMTOKEN attribute. Each NMTOKEN value is separated by white space

Entity Declarations

An entity declaration works similarly to the `define` command in many programming languages, PHP included. I briefly introduced entity references in the preceding section, “An Introduction to XML Syntax.” To recap, an entity reference acts as a substitute for another piece of content. When the XML document is parsed, all occurrences of this entity are replaced with the content that it represents. There are two types of entities: internal and external.

Internal Entities

Internal entities are used much like string variables are, correlating a name with a piece of text. For example, if you wanted to associate a name that pointed to your company's copyright statement you would declare the entity as follows:

```
<!ENTITY Copyright "Copyright 2000 YourCompanyName. All Rights Reserved.">
```

When the document is parsed, all occurrences of &Copyright are replaced with "Copyright 2000 YourCompanyName. All Rights Reserved." Any XML in the replacement content would be parsed as if it had originally appeared in the document!

An internal reference works fine when you plan on using an entity for a specific or limited number of XML documents. However, if your company is processing quite a few XML documents, then perhaps using an external entity is your best bet.

External Entities

External entities also can be used to reference content in another file. This entity type can reference text, but it can also reference binary data, such as a graphic. Referring back to the previous copyright example, you may want to store this information in another file to facilitate its later modification. You could declare an external entity pointing to it as follows:

```
<!ENTITY Copyright SYSTEM http://yoursite.com/administration/copyright.xml">
```

When the XML document is later parsed, any references to &Copyright; will be substituted with the content in the copyright.xml document. This information will be parsed just as if it originally appeared in the document.

It is also useful to use external entities to point to graphics. For example, if you wanted to place a logo in certain XML documents, you could declare an external entity pointing to it, as shown here:

```
<!ENTITY food_picture SYSTEM http://yoursite.com/food/logo.gif>
```

Just as is the case with the copyright example, any reference to `&food_picture` will be replaced with the graphic to which the external entity points. However, since this data is binary and not text, it will not be parsed.

XML References

Although the preceding XML introduction is sufficient for understanding the basic framework of XML documents, there is still quite a bit more to be learned. The following links point to some of the more comprehensive XML resources available on the Internet:

- `http://www.w3.org/XML/`
- `http://www.xml.com/pub/ArticlesbyTopic`
- `http://www.ibm.com/developer/xml/`
- `http://www.oasis-open.org/cover/`

The remainder of this chapter is devoted to how PHP can be used to parse XML documents. Although it seems complicated (parsing any type of document can be a daunting task), I think you'll be rather surprised at how easy it is once you've learned the basic strategy used by PHP for doing so.

PHP and XML

PHP's XML functionality is implemented using James Clark's Expat (XML Parser Toolkit) package, at http://www.jclark.com/xml/. Expat comes packaged with Apache 1.3.7 and later, so you won't need to specifically download it if you are using a recent version of Apache. To use PHP's XML functionality, you'll need to configure PHP using `-with-xml`.

NOTE *Expat 2.0 is currently being developed by Clark Cooper. More information is at http://expat.sourceforge.net/.*

Although at first the idea of parsing XML data using PHP (or any language) seems intimidating, much of the work is already done for you by PHP's predefined functionality. All that you are left to do is define new functions tailored to your own DTD definitions and then apply these functions to PHP's easy-to-follow XML parsing process.

Before I begin introducing PHP's XML function set, take a moment to reconsider the very basic pieces that comprise an XML document. This will help you understand the mechanics behind why certain functions are an indispensable part of any XML parser. On the most general level, there are nine components of an XML document:

- Opening tags
- Attributes
- Character data
- Closing tags
- Processing instructions
- Notation declarations
- External entity references
- Unparsed entities
- Other components (comments, XML declaration, etc.)

Given these nine components, in order to effectively parse an XML document, functions need to be defined that handle each of these components. Once they are defined, you use PHP's various predefined callback functions that act to integrate your custom handler functions into the overall XML parsing process. You can think of PHP's general XML parsing process as a series of five steps:

1. Create your customer handler functions. Of course, if you intend on working with XML documents in a consistent fashion, you will only need to create these functions once and subsequently concentrate on maintaining them.

2. Create the XML parser that will be used to parse the document. This is accomplished by calling `xml_parser_create()`.

3. Use the predefined callback functions to register your handler functions with the XML parser.

4. Open the XML file, read the data contained in it, and pass this data to the XML parser. Note that to parse the data, you only need to call `xml_parse()`! This function is responsible for implicitly calling all of the previously defined handler functions.

5. Free up the XML parser, essentially clearing the data from it. This is accomplished by calling `xml_parser_free()`.

The purpose of each of these steps will become apparent as you read the next section, "PHP's Handler Functions."

PHP's Handler Functions

There are eight predefined set functions that act to register the functions that will be used to handle the various components of an XML document:

Keep in mind that you *must* define the functions that will be tied into the handler functions; otherwise an error will occur. Each predefined register function and the specifications for the corresponding handler functions are presented in this section.

xml_set_character_data_handler()

This function registers the handler function that works with character data. Its syntax is:

```
int xml_set_character_data_handler(int parser, string characterHandler)
```

The input parameter `parser` refers to the XML parser handler. The input parameter `characterHandler` refers to the name of the function created to handle the character data. The function specified by `characterHandler` is defined here:

```
function characterHandler(int parser, string data) {
…
}
```

The input parameter `parser` refers to the XML parser handler, and `data` to the character data that has been parsed.

xml_set_default_handler()

This function specifies the handler function that is used for all components of the XML document that do not need to be registered. Examples of these components include the XML declaration and comments. Its syntax is:

```
int xml_set_default_handler(int parser, string defaultHandler)
```

The input parameter `parser` refers to the XML parser handler. The input parameter `defaultHandler` refers to the name of the function created to handle the XML element. The function specified by `defaultHandler` is defined here:

```
function defaultHandler(int parser, string data) {
…
}
```

The input parameter `parser` refers to the XML parser handler, and `data` to the character data that will be handled by default.

xml_set_element_handler()

This function registers the handler functions that work with the parse starting and ending element tags. Its syntax is:

```
int xml_set_element_handler(int parser, string startTagHandler, string
endTagHandler)
```

The input parameter `parser` refers to the XML parser handler. The input parameters `startTagHandler` and `endTagHandler` refer to the names of the functions created to handle the starting and ending tag elements, respectively. The function specified by `startTagHandler` is defined as:

```
function startTagHandler(int parser, string tagName, string attributes[]) {
…
}
```

The input parameter `parser` refers to the XML parser handler, `tagName` to the name of the opening tag element being parsed, and `attributes` to the array of attributes that may accompany the tag element.

The function specified by `endTagHandler` is defined as:

```
function endTagHandler(int parser, string tagName) {
…
}
```

The input parameter `parser` refers to the XML parser handler, `tagName` to the name of the closing tag element being parsed.

xml_set_external_entity_ref_handler()

This function registers the handler function that works with external entity references. Its syntax is:

```
int xml_set_external_entity_ref_handler(int parser, string externalHandler)
```

The input parameter parser refers to the XML parser handler. The input parameter externalHandler refers to the name of the function created to handle the external entity. The function specified by externalHandler is defined here:

```
function externalHandler(int parser, string entityReference, string base, string
systemID, string publicID) {
…
}
```

The input parameter parser refers to the XML parser handler, entityReference to the name of the entity reference, systemID to the system identifier of the entity reference, and publicID to the public identifier of the entity reference. The parameter base is currently not used by the function, but needs to be declared anyway.

xml_set_notation_declaration_handler()

This function registers the handler function that works with notation declarations. Its syntax is:

```
int xml_set_notation_declaration_handler(int parser, string notationHandler)
```

The input parameter parser refers to the XML parser handler. The input parameter notationHandler refers to the name of the function created to handle the notation declaration. The function specified by notationHandler is defined here:

```
function notationHandler(int parser, string notationDeclaration, string base,
string systemID, string publicID) {
…
}
```

The input parameter parser refers to the XML parser handler, notationDeclaration to the name of the notation declaration, systemID to the system identifier of the notation declaration, and publicID to the public identifier of the notation declaration. The parameter base is currently not used by the function, but needs to be declared anyway.

xml_set_object()

This function makes it possible to use the XML parser from within an object. Its syntax is:

```
void xml_set_object(int parser, object &object)
```

The input parameter `parser` refers to the XML parser handler, and the `object` reference refers to the object containing the methods used to handle the XML components. The specific purpose of this function is to identify the `parser` with that specific `object`. Typically you'll use this function in an object's constructor method, following it with the various handler function definitions:

```
class xmlDB {
VAR $xmlparser;

     function xmlDB() {
          $this->xmlparser = xml_parser_create();
          // associate the parser with the object
          xml_set_object($this->xmlparser,&$this);
          // define the callback functions
          xml_set_element_handler($this->xmlparser,"startTag","endTag");
          xml_set_character_data_handler($this->xmlparser,"characterData");
     }

. . . The handler functions startTag, endTag, characterData and others are created
here

} // end class xmlDB
```

As an exercise, try commenting out the call to xml_set_object(). You'll see that subsequent execution results in error messages regarding the inability to call the handler methods belonging to the object.

xml_set_processing_instruction_handler()

This function registers the handler function that works with processing instructions. Its syntax is:

```
xml_set_processing_instruction_handler(int parser, string processingIntHandler)
```

The input parameter `parser` refers to the XML parser handler. The input parameter `processingHandler` refers to the name of the function created to handle the processing instruction. The function specified by `processingIntHandler` is defined here:

```
function processingIntHandler(int parser, string processingApp, string
instruction) {

…
}
```

The input parameter `parser` refers to the XML parser handler, `processingApp` to the name of the application that should process the instruction, and `instruction` to the instruction that is passed to the application.

xml_set_unparsed_entity_decl_handler()

This function registers the handler function that works with external entity references. Its syntax is:

```
int xml_set_unparsed_entity_decl_handler(int parser, string unparsedEntityHandler)
```

The input parameter `parser` refers to the XML parser handler. The input parameter `unparsedEntityHandler` refers to the name of the function created to handle the unparsed entity. The function specified by `unparsedEntityHandler` is defined here:

```
function unparsedEntityHandler(int parser, string entDec, string base, string
sysID, string pubID, string NName) {
…
}
```

The input parameter `parser` refers to the XML parser handler, `entDec` to the name of the entity being defined, `sysID` to the system identifier of the notation declaration, and `pubID` to the public identifier of the notation declaration. The parameter `base` is currently not used by the function, but needs to be declared anyway. Finally, `NName` refers to the name of the notation declaration.

This concludes the introduction of the register and handler functions. However, these are not the only functions you need to effectively parse XML documents. The remainder of PHP's predefined XML functionality is presented next.

PHP's Parsing Functions

While it is not necessary to implement each one of PHP's handler functions (an XML document does not have to use every type of element), there are three functions that should be in every parsing script. These functions are described below.

xml_parser_create()

Before parsing an XML document, you must first create a parser. The syntax for doing so is:

```
int xml_parser_create([string encoding])
```

The optional input parameter encoding can be used to specify the source encoding. Currently, there are three supported source encodings:

- UTF-8
- US-ASCII
- ISO-8859-1 (Default)

Much like `fopen()` returns a handle to an opened file, `xml_parser_create()` returns a parser handle. This handle will then be passed into the various other functions throughout the parsing process. If you are simultaneously parsing several documents, you can also define multiple parsers.

xml_parse()

This function does the actual parsing of the document. Its syntax is:

```
int xml_parse(int parser, string data [int isFinal])
```

The parameter `parser` specifies which XML parser to use. This is the variable returned `by xml_parser_create()`. The optional input parameter `isFinal`, when set and true, tells the parser to stop. Typically this would be when the end of the file being parsed is reached.

xml_parser_free()

This function frees the resources devoted to the parser. Its syntax is:

```
int xml_parser_free(int parser)
```

The input parameter `parser` refers to the XML parser handler.

Useful Functions

PHP also offers a number of other functions that can further facilitate XML parsing. These functions are presented here.

utf8_decode()

This function will convert data to ISO-8859-1 encoding. It is assumed that the data being converted is of the UTF-8 encoding format. Its syntax is:

```
string utf8_decode(string data)
```

The input parameter `data` refers to the UTF-8-encoded data that is to be converted.

utf8_encode()

This function will convert data from the ISO-8859-1 encoding format to the UTF-8 encoding format. Its syntax is:

```
string utf8_encode(string data)
```

The input parameter `data` refers to the ISO-8859-1-encoded data that is to be converted.

xml_get_error_code()

The function `xml_get_error_code()` retrieves the error value specific to an XML parsing error. This can then be passed to `xml_error_string()` (introduced next) for interpretation. Its syntax is:

```
int xml_error_code(int parser)
```

The input parameter `parser` refers to the XML parser handler. An example of usage is shown below, in the introduction to the function `xml_get_current_line_number()`.

xml_error_string()

When a parsing error occurs, it is assigned an error code. The function `xml_error_string()` can be passed this code, returning the text description of the code. Its syntax is:

```
string xml_error_string(int code)
```

The input parameter code refers to the error code assigned to the respective error. This error code can be retrieved from the function `xml_get_error_code()`. An example of usage is shown below, in the introduction to the function `xml_get_current_line_number()`.

xml_get_current_line_number()

This function retrieves the line currently being parsed by the XML parser. Its syntax is:

```
int get_current_line_number(int parser)
```

The input parameter `parser` refers to the XML parser handler. An example follows:

```
while ($line = fread($fh, 4096)) :
     if (! xml_parse($xml_parser, $line, feof($fh))) :
          $err_string = xml_error_string(xml_get_error_code($xml_parser));
          $line_number = xml_get_current_line_number($xml_parser);
          print "Error! [Line $line_number]: $err_string";
     endif;
endwhile;
```

If a parsing error occurred in line six of the file pointed to by `$fh`, you would see an error message similar to the following in the parsed output:

```
Error! [Line 6]: mismatched tag
```

xml_get_current_column_number()

The function `xml_get_current_column_number()` can be used in conjunction with `xml_get_current_line_number()` to pinpoint the exact location of an error in an XML document. Its syntax is:

```
int get_current_column_number(int parser)
```

The input parameter `parser` refers to the XML parser handler. Reconsider the previous example:

```
while ($line = fread($fh, 4096)) :
     if (! xml_parse($xml_parser, $line, feof($fh))) :
          $err_string = xml_error_string(xml_get_error_code($xml_parser));
          $line_number = xml_get_current_line_number($xml_parser);
          $column_number = xml_get_current_column_number($xml_parser);
          print "Error! [Line $line_number, Column $column_number]: $err_string";
     endif;
endwhile;
```

If a parsing error occurred in line six of the file pointed to by `$fh`, you would see an error message similar to the following in the parsed output:

```
Error! [Line 6 Column 2]: mismatched tag
```

XML Parser Options

PHP currently offers two parser options:

- XML_OPTION_CASE_FOLDING, which is nothing more than converting tag element names to uppercase.
- XML_OPTION_TARGET_ENCODING, which specifies the document encoding output by the XML parser. Currently, UTF-8, ISO-8859-1, and US-ASCII encoding support is available.

These options can be both retrieved and modified using the functions `xml_parser_get_option()` and `xml_parser_set_option()`, respectively.

xml_parser_get_option()

The function `xml_parser_get_option()` retrieves the XML parser's options. Its syntax is:

```
int xml_parser_get_option(int parser, int option)
```

The input parameter `parser` refers to the XML parser handler. The parameter `option` specifies the option that will be retrieved, its value specified by the parameter `value`. An example follows:

```
$setting = xml_parser_get_option($xml_parser, XML_OPTION_CASE_FOLDING);
print "Case Folding: $setting";
```

Assuming that the XML_OPTION_CASE_FOLDING option has not been already explicitly modified, its default option of enabled will be retrieved. Therefore, executing this code would result in the outcome:

```
CASE FOLDING: 1
```

xml_parser_set_option()

The function `xml_parser_set_option()` configures the XML parser's options. Its syntax is:

```
int xml_parser_set_option(int parser, int option, mixed value)
```

The input parameter `parser` refers to the XML parser handler. The parameter `option` specifies the option that will be set, its value specified by the parameter `value`. An example follows:

```
xml_parser_set_option($xml_parser, XML_OPTION_TARGET_ENCODING, "UTF-8");
```

Execution of this command changes the target encoding option from the default of ISO-8859-1 to UTF-8.

XML-to-HTML Conversion

Suppose that you had an XML document containing a list of bookmarks, entitled bookmarks.xml. It looks similar to the following:

```
<?xml version="1.0"?>
<website>
<title>Epicurious</title>
<url>http://www.epicurious.com</url>
<description>
Epicurious is a great online cooking resource, providing tutorials, recipes,
forums and more.
</description>
</website>
```

Now assume that you wanted to parse bookmarks.xml, displaying its contents in a format readable from within a PC browser. Listing 14-3 will parse this file and reformat as necessary.

Listing 14-3: XML-to-HTML conversion parser

```
<?

class XMLHTML {
     VAR $xmlparser;
     VAR $tagcolor = "#800000";
     VAR $datacolor = "#0000ff";

     function XMLHTML() {
          $this->xmlparser = xml_parser_create();
          xml_set_object($this->xmlparser, &$this);
          xml_set_element_handler($this->xmlparser, "startTag", "endTag");
          xml_set_character_data_handler($this->xmlparser, "characterData");
     }

     // This function is responsible for handling all starting element tags.
     function startTag($parser, $tagname, $attributes) {
          GLOBAL $tagcolor;
          print "<font size=\"-2\" color=\"$this->tagcolor\" face=\"arial,
          verdana\">&lt;$tagname&gt;</font> <br>";
     }

     // This function is responsible for handling all character data.
     function characterData($parser, $characterData) {
          GLOBAL $datacolor;
          print "<font size=\"-2\" color=\"$this->datacolor\" face=\"arial,
          verdana\">&nbsp;&nbsp;&nbsp;$characterData</font> <br>";
     }

     // This function is responsible for handling all ending element tags.
     function endTag($parser, $tagname) {
          GLOBAL $tagcolor;
          print "<font size=\"-2\" color=\"$this->tagcolor\" face=\"arial,
          verdana\">&lt;/$tagname&gt;</font> <br>";
     }

     function parse($fp) {
          // xml_parse($this->xmlparser,$data);
          // Parse the XML file
          while ( $line = fread($fp, 4096) ) :
              // If something goes wrong, stop and print an error message.
                   if ( ! xml_parse($this->xmlparser, $line, feof($fp))) :
                   die(sprintf("XML error: %s at line %d",
                   xml_error_string(xml_get_error_code($this->xmlparser)),
                   xml_get_current_line_number($this->xmlparser)));
              endif;
         endwhile;
    }

} // end class

// Open the XML file for parsing
$xml_file = "bookmarks.xml";
$fp = fopen($xml_file, "r");

// create new object
$xml_parser = new XMLHTML;

// parse $xml_file
$xml_parser->parse($fp);

?>
```

Once bookmarks.xml is parsed, you would see it displayed in the browser as shown below.

```
<WEBSITE>
<TITLE>
   Epicurious
</TITLE>
<URL>
   http://www.epicurious.com
</URL>
<DESCRIPTION>
   Epicurious is a great online cooking resource, providing tutorials, recipes,
forums and more.
</DESCRIPTION>
</WEBSITE>
```

Of course, this doesn't accomplish too much; it merely makes the XML viewable within the browser. With just a few modifications to Listing 14-3, you could begin parsing links to ensure that they are displayed as working hyperlinks, convert the data found within the <TITLE>…</TITLE> tags to boldface, etc. As you can see, I also declared two font colors as object attributes to show that you can easily format the data being output to the browser.

A Final Note About PHP and XML

Throughout this chapter I introduced XML and the various functions that PHP uses to parse XML documents. However, as it applies to PHP, I've only actually covered one of the three specifications that define XML and did not delve into issues regarding XSL or XLL. Of course, to truly take advantage of separating content from presentation, all three of these components need to be fully exploited, or at the very least, XML and XSL.

Unfortunately, at the time of this writing, PHP does not provide a complete solution for those wishing to work with XML using PHP as the sole handling language. Of course, as PHP's capabilities continue to expand, I'm fairly confident that these issues will be resolved.

NOTE *One particularly promising development in this area is an XSLT (XSL transformation) processor named Sablotron, developed by Ginger Alliance Ltd. (http://www.gingerall.com). On October 12, 2000, it was announced that PHP 4.03 is now available with the Sablotron module extension on both the Linux and Windows platforms. Be sure to check this out for further developments.*

What's Next?

This chapter covered quite a bit of ground regarding XML and PHP's XML parsing functionality. I began with a brief synopsis of the history of markup languages and subsequently introduced you to XML, its advantages, and a primer of its syntactical constructs. The remainder of the chapter was devoted to introducing the many predefined XML functions offered by PHP, finally concluding with several examples of how PHP can be used to parse and output XML data. In particular, the following topics were covered:

- A brief introduction to markup languages
- SGML
- An introduction to XML
- XML syntax

- The document type definition (DTD)
- PHP and XML

Chapter 15 switches gears, covering two prominent technologies, namely, JavaScript and the Component Object Model (COM), and how PHP can interact with them.