A Really, Really, Really Good Introduction to XML

Aug. 24, 2005 • 106 min read • original

In this chapter, we'll cover the basics of XML – essentially, most of the information you'll need to know to get a handle on this exciting technology. After we're done exploring some terminology and examples, we'll jump right in and start working with XML documents. Then, we'll spend some time starting the project we'll develop through the course of this book: building an XML-powered content management system.

This excerpt is taken from No Nonsense XML Web Development with PHP, SitePoint's new release, by Thomas Myer, which was designed to help you start using XML to build intelligent 'Future-Proof' PHP applications today.

The title contains over 350 pages of XML and PHP goodies. It walks you through the process of building a fully-functional XML-based content management system with PHP. And all the code used in the book is available to customers in a downloadalbe archive.

To find out more about "No Nonsense XML Web Development with PHP", visit the book's information page, or review the contents of the entire publication. As always, you can download this excerpt as a PDF if you prefer.

Chapter 1. Introduction to XML

Who here has heard of XML? Okay, just about everybody. If ever there were a candidate for "Most Hyped Technology" during the late 90s and the current decade, it's XML (though Java would be a close contender for the title).

Whenever I talk about XML with developers, designers, technical writers, or other Web professionals, the most common question I'm asked is, "What's the big deal?" In this book, I'll explain exactly what the big deal is – how XML can be used to make your Web applications smarter, more versatile, and more powerful. I'll try to stay away from the grandstanding hoopla that has characterized much of the discussion of XML; instead, I'll give you the background and know-how you'll need to make XML a part of your professional skillset.

What is XML?

So, what is XML? Whenever a group of people asks this question, I always look at the individuals' body language. A significant portion of the group leans forward eagerly, wanting to learn more. The others either roll their eyes in anticipation of hype and half-formed theories, or cringe in fear of a long, dry history of markup languages. As a result, I've learned to keep my explanation brief.

The essence of XML is in its name: Extensible Markup Language.

Extensible

XML is extensible. It lets you define your own tags, the order in which they occur, and how they should be processed or displayed. Another way to think about extensibility is to consider that XML allows all of us to extend our notion of what a document is: it can be a file that lives on a file server, or it can be a transient piece of data that flows between two computer systems (as in the case of Web Services).

Markup

The most recognizable feature of XML is its tags, or elements (to be more accurate). In fact, the elements you'll create in XML will be very similar to the elements you've already been creating in your HTML documents. However, XML allows you to define your own set of tags.

Language

XML is a language that's very similar to HTML. It's much more flexible than HTML because it allows you to create your own custom tags. However, it's important to realize that XML is not just a language. XML is a meta-language: a language that allows us to create or define other languages. For example, with XML we can create other languages, such as RSS, MathML (a mathematical markup language), and even tools like XSLT. More on this later.

Why Do We Need XML?

Okay, we know what it is, but why do we need XML? We need it because HTML is specifically designed to describe documents for display in a Web browser, and not much else. It becomes cumbersome if you want to display documents in a mobile device or do anything that's even slightly complicated, such as translating the content from German to English. HTML's sole purpose is to allow anyone to quickly create Web documents that can be shared with other people. XML, on the other hand, isn't just suited to the Web – it can be used in a variety of different contexts, some of which may not have anything to do with humans interacting with content (for example, Web Services use XML to send requests and responses back and forth).

HTML rarely (if ever) provides information about how the document is structured or what it means. In layman's terms, HTML is a presentation language, whereas XML is a data-description language.

For example, if you were to go to any ecommerce Website and download a product listing, you'd probably get something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>ABC Products</title>
<meta http-equiv="Content-Type"
   content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1>ABC Products</h1>
<h2>Product One</h2>
<p>Product One is an exciting new widget that will simplify your
 life.</p>
<p><b>Cost: $19.95</b></p>
<p><b>Shipping: $2.95</b></p>
<h2>Product Two</h2>
...
<h3>Product Three</h3>
<p><i>Cost: $24.95</i></p>
<p>This is such a terrific widget that you will most certainly
 want to buy one for your home and another one for your
 office!</p>
...
</body>
</html>
```

Take a good look at this – admittedly simple – code sample from a computer’s perspective. A human can certainly read this document and make the necessary semantic leaps to understand it, but a computer couldn’t.

Semantics and Other Jargon

You’re going to be hearing a lot of talk about “semantics” and other linguistics terms in this chapter. It’s unavoidable, so bear with me. Semantics is the study of meaning in language.

Humans are much better at semantics than computers, because humans are really good at deriving meaning. For example, if I asked you to list as many names for “female animals” as you could, you’d probably start with “lioness”, “tigress”, “ewe”, “doe” and so on. If you were presented with a list of these names and asked to provide a category that contained them all, it’s likely you’d say something like “female animals.” Furthermore, if I asked you what a lioness was, you’d say, “female lion.”

If I further asked you to list associated words, you might say “pride,” “hunt,” “savannah,” “Africa,” and the like. From there, you could make the leap to other wild cats, then to house cats and maybe even dogs (cats and dogs are both pets, after all). With very little effort, you’d be able to build a stunning semantic landscape, as it were.

Needless to say, computers are really bad at this game, which is a shame, as many computing tasks require semantic skill. That’s why we need to give computers as much help as we can.

For example, a human can probably deduce that the `<h2>` tag in the above document has been used to tag a product name within a product listing. Furthermore, a human might be able to guess that the first paragraph after an `<h2>` holds the description, and that the next two paragraphs contain price and shipping information, in bold.

However, even a cursory glance at the rest of the document reveals some very human errors. For example, the last product name is encapsulated in `<h3>` tags, not `<h2>` tags. This last product listing also displays a price before the description, and the price is italicized instead of appearing in bold.

A computer program (and even some humans) that tried to decipher this document wouldn't be able to make the kinds of semantic leaps required to make sense of it. The computer would be able only to render the document to a browser with the styles associated with each tag. HTML is chiefly a set of instructions for rendering documents inside a Web browser; it's not a method of structuring documents to bring out their meaning.

If the above document were created in XML, it might look a little like this:

```
<?xml version="1.0"?>
 <productListing title="ABC Products">
  <product>
    <name>Product One</name>
    <description>Product One is an exciting new widget that will
      simplify your life.</description>
    <cost>$19.95</cost>
    <shipping>$2.95</shipping>
  </product>
  <product>
    <name>Product Two</name>
    ...
  </product>
  <product>
    <name>Product Three</name>
    <description>This is such a terrific widget that you will
      most certainly want to buy one for your home and another one
      for your office!</p>
    <cost>$24.95</cost>
    <shipping>$0.00</shipping>
  </product>
  ...
 </productListing>
```

Notice that this new document contains absolutely no information about display. What does a `<product>` tag look like in a browser? Beats me – we haven't defined that yet. Later on, we'll see how you can use technologies like CSS and XSLT to transform your XML into any format you like. Essentially, XML allows you to *separate information from presentation* – just one of its many powerful abilities.

When we concentrate on a document's structure, as we've done here, we are better able to ensure that our information is correct. In theory, we should be able to look at any XML document and understand instantly what's going on. In the example above, we know that a product listing contains products, and that each product has a name, a description, a price, and a shipping cost. You could say, rightly, that each XML document is *self-describing*, and is readable by both humans and software.

Now, everyone makes mistakes, and XML programmers are no exception. Imagine that you start to share your XML documents with another developer or company, and, somewhere along the line, someone places a product's description after its price. Normally, this wouldn't be a big deal, but perhaps your Web application requires that the description appears after the product name *every* time.

To ensure that everyone plays by the rules, you need a *DTD* (a document type definition), or schema. Basically, a DTD provides instructions about the structure of your particular XML document. It's a lot like a rule book that states which tags are legal, and where. Once you have a DTD in place, anyone who creates product listings for your application will have to follow the rules. We'll get into DTDs a little later. For now, though, let's continue with the basics.

A Closer Look at the XML Example

From the casual observer's viewpoint, a given XML document, such as the one we saw in the previous section, appears to be no more than a bunch of tags and letters. But there's more to it than that!

A Structural Viewpoint

Let's consider our XML example from a structural standpoint. No, not the kind of structure we bring to a document by marking it up with XML tags; let's look at this example on a more granular level. I want to examine the contents of a typical XML file, character by character.

The simplest XML elements contain an opening tag, a closing tag, and some content. The opening tag begins with a left angle bracket (`<`), followed by an element name that contains letters and numbers (but no spaces), and finishes with a right angle bracket (`>`). In XML, content is usually parsed character data. It could consist of plain text, other XML elements, and more exotic things like XML entities, comments, and processing instructions (all of which we'll see later). Following the content is the closing tag, which exhibits the same spelling and capitalization as your opening tag, but with one tiny change: a `/` appears right before the element name.

Here are a few examples of valid XML elements:

```
<myElement>some content here</myElement>
 <elements>
  <myelement>one</myelement>
  <myelement>two</myelement>
 </elements>
```

Elements, Tags, or Nodes?

I'll refer to XML elements, XML tags, and XML nodes at different points in this book. What's the deal? Well, for the layman, these terms are interchangeable, but if you want to get technical (and who'd want to do that in a technical book?) each has a very precise meaning:

- An element consists of an opening tag, its attributes, any content, and a closing tag.
- A tag – either opening or closing – is used to mark the start or end of an element.
- A node is a part of the hierarchical structure that makes up an XML document. "Node" is a generic term that applies to any type of XML document object, including elements, attributes, comments, processing instructions, and plain text.

If you're used to working with HTML, you've probably created many documents that are missing end tags, use different capitalization in opening and closing tags, and contain improperly nested tags.

You won't be able to get away with any of that in XML! In this language, the `<myElement>` tag is different from the `<MYELEMENT>` tag, and both are different from the `<myELEMENT>` tag. If your opening tag is `<myELEMENT>` and your closing tag is `</Myelement>`, your document won't be valid.

If you use attributes on any elements, then attribute values must be single- or double-quoted. No longer can you get by with bare attribute values like you did in HTML! Let's see an example. The following is okay in HTML:

```
<h1 class=topHeader>
```

In XML, you'd have to put quotes (either single or double) around the attribute value, like this:

```
<h1 class="topHeader">
```

Also, if you nest your elements improperly (i.e. close an element before closing another element that is inside it), your document won't be valid. (I know I keep mentioning validity – we'll talk about it in detail soon!) For example, Web browsers don't generally complain about the following:

```
<b>Some text that is bolded, some that is <i>italicized</b></i>.
```

In XML, this improper nesting of elements would cause the program reading the document to raise an error.

As XML allows you to create any language you want, the inventors of XML had to institute a special rule, which happens to be closely related to the proper nesting rule. The rule states that each XML document must contain a single root element in which all the document's other elements are contained. As we'll see later, almost every single piece of XML development you'll do is facilitated by this one simple rule.

Attributes

Did you notice the `<productListing>` opening tag in our example? Inside the tag, following the element name, was the data `title="ABC Products"`. This is called an attribute.

You can think of attributes as adjectives – they provide additional information about the element that may not make any sense as content. If you've worked with HTML, you're familiar with such attributes as the `src` (file source) on the `<img>` tag.

What information should be contained in an attribute? What should appear between the tags of an element? This is a subject of much debate, but don't worry, there really are no wrong answers here. Remember: you're the one defining your own language. Some developers (including me!) apply this rule of thumb: use attributes to store data that doesn't necessarily need to be displayed to a user of the information. Another common rule of thumb is to consider the length of the data. Potentially large data should be placed inside a tag; shorter data can be placed in an attribute. Typically, attributes are used to "embellish" the data contained within the tag.

Let's examine this issue a little more closely. Let's say that you wanted to create an XML document to keep track of your DVD collection. Here's a short snippet of the code you might use:

```
<dvdCollection>
  <dvd>
    <id>1</id>
    <title>Raiders of the Lost Ark</title>
    <release-year>1981</release-year>
    <director>Steven Spielberg</director>
    <actors>
      <actor>Harrison Ford</actor>
      <actor>Karen Allen</actor>
      <actor>John Rhys-Davies</actor>
    </actors>
  </dvd>
  ....
</dvdCollection>
```

It's unlikely that anyone who reads this document would need to know the ID of any of the DVDs in your collection. So, we could safely store the ID as an attribute of the `<dvd>` element instead, like this:

```
<dvd id="1">
```

In other parts of our DVD listing, the information seems a little bare. For instance, we're only displaying an actor's name between the `<actor>` tags – we could include much more information here. One way to do so is with the addition of attributes:

```
<actor type="superstar" gender="male" age="50">Harrison Ford
```

</actor>

In this case, though, I'd probably revert to our rule of thumb – most users would probably want to know at least some of this information. So, let's convert some of these attributes to elements:

```
<actor type="superstar">
```

<name>Harrison Ford</name>

<gender>male</gender>

<age>50</age>

</actor>

Beware of Redundant Data

From a completely different perspective, one could argue that you shouldn't have all this repetitive information in your XML file. For example, your collection's bound to include at least one other movie that stars Harrison Ford. It would be smarter, from an architectural point of view, to have a separate listing of actors with unique IDs to which you could link. We'll discuss these questions at length throughout this book.

Empty-Element Tags

Some XML elements are said to be empty – they contain no content whatsoever. Familiar examples are the `img` and `br` elements in HTML. In the case of `img`, for example, all the element's information is contained in its tag's attributes. The `<br>` tag, on the other hand, does not normally contain any attributes – it just signifies a line break.

Remember that in XML all opening tags must be matched by a closing tag. For empty elements, you can use a single empty-element tag to replace this:

```
<myEmptyElement></myEmptyElement>
```

with this:

```
<myEmptyElement/>
```

The `/` at the end of this tag basically tells the parser that the element starts and ends right here. It's an efficient shorthand method that you can use to mark up empty elements quickly.

The XML Declaration

The line right at the top of our example is called the XML declaration:

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

It's not strictly necessary to include this line, but it's the best way to make sure that any device that reads the document will know that it's an XML document, and to which version of XML it conforms.

Entities

I mentioned entities earlier. An entity is a handy construct that, at its simplest, allows you to define special characters for insertion into your documents. If you've worked with HTML, you know that the `&lt;` entity inserts a literal `<` character into a document. You can't use the actual character because it would be treated as the start of a tag, so you replace it with the appropriate entity instead.

XML, true to its extensible nature, allows you to create your own entities. Let's say that your company's copyright notice has to go on every single document. Instead of typing this notice over and over again, you could create an entity reference called `copyright_notice` with the proper text, then use it in your XML documents as `&copyright_notice;` . What a time-saver!

We'll cover entities in more detail later on.

More than Structure...

XML documents are more then just a sequence of elements. If you take another, closer look at our product or DVD listing examples, you'll notice two things:

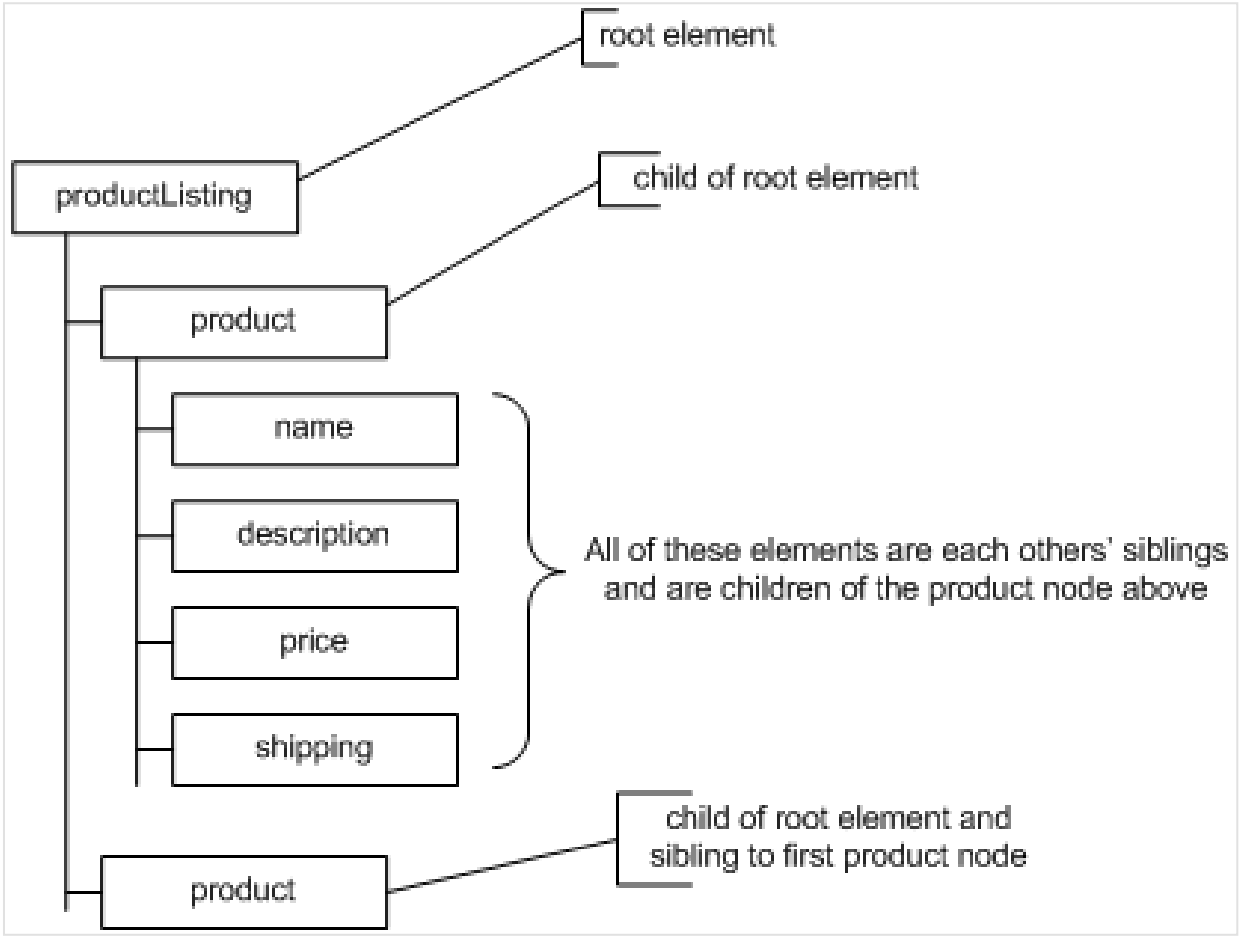
- The documents are self-describing, as we've already discussed.
- The documents are really a hierarchy of nested objects.

Let's elaborate on the first point very quickly. We've already said that most (if not all) XML documents are self-describing. This feature, combined with all that content encapsulated in opening and closing tags, takes all XML documents far past the realm of mere data and into the revered halls of *information*.

Data can comprise a string of characters or numbers, such as `5551238888` . This string can represent anything from a laptop's serial number, to a pharmacy's prescription ID, to a phone number in the United States. But the only way to turn this data into information (and therefore make it useful) is to add context to it – once you have context, you can be sure about what the data represents. In short, `<phone country="us">5551238888</phone>` leaves no doubt that this seemingly arbitrary string of numbers is in fact a U.S. phone number.

When you take into account the second point – that an XML document is really a hierarchy of objects – all sorts of possibilities open up. Remember what we discussed before – that, in an XML document, one element contains all the others? Well, that root element becomes the root of our hierarchical tree. You can think of that tree as a family tree, with the root element having various children (in this case, product elements), and each of those having various children (name, description, and so on). In turn, each product element has various siblings (other product elements) and a parent (the root), as shown in Figure 1.1, "The logical structure of an XML document.".

Figure 1.1. The logical structure of an XML document.


root element
productListing
child of root element
product
name
description
price
shipping
All of these elements are each others' siblings and are children of the product node above
child of root element and sibling to first product node
product

Because what we have is a tree, we should be able to travel up and down it, and from side to side, with relative ease. From a programmatic stance, most of your work with XML will focus on properly creating and navigating XML structures.

There's one final point about hierarchical trees that you should note. Before, we talked about transforming data into information by adding context. Well, when we start building hierarchies of information that indicate natural relationships (known as *taxonomies*), we've just taken the first giant leap toward turning information into knowledge. That statement itself could spawn a whole other book, so we'll just have to leave it at that and move on!

Formatting Issues

Earlier in this chapter, I made a point about XML allowing you to separate information from presentation. I also mentioned that you could use other technologies, like CSS (Cascading Style Sheets) and XSLT (Extensible Stylesheet Language Transformations), to make the information display in different contexts.

Note

Notice that in XSLT, it's "stylesheet," but in CSS it's "style sheet"! For the sake of consistency, we'll call them all "style sheets" in this book.

In later chapters, I'll go into plenty of detail on both CSS and XSLT, but I wanted to make a brief point here. Because we've taken the time to create XML documents, our information is no longer locked up inside proprietary formats such as word processors or spreadsheets. Furthermore, it no longer has to be "re-created" every time you want to create alternate displays of that information: all you have to do is create a style sheet or transformation to make your XML presentable in a given medium.

For example, if you stored your information in a word processing program, it would contain all kinds of information about the way it should appear on the printed page – lots of bolding, font sizes, and tables. Unfortunately, if that document also had to be posted to the Web as an HTML document, someone would have to convert it (either manually or via software), clean it up, and test it. Then, if someone else made changes to the original document, those changes wouldn't cascade to the HTML version. If yet another person wanted to take the same information and use it in a slide presentation, they might run the risk of using outdated information from the HTML version. Even if they did get the right information into their presentation, you'd still need to track three locations in which your information lived. As you can see, it can get pretty messy!

Now, if the same information were stored in XML, you could create three different XSLT files to transform the XML into HTML, a slide presentation, and a printer-friendly file format such as PostScript. If you made changes to the XML file, the other files would also change automatically once you passed the XML file through the process. (This notion, by the way, is an essential component of single-sourcing – i.e. having a "single source" for any given information that's reused in another application.)

As you can see, separating information from presentation makes your XML documents reusable, and can save hassles and headaches in environments in which a lot of information needs to be stored, processed, handled, and exchanged.

Here's another example. This book will actually be stored as XML (in the DocBook schema). That means the publisher can generate sample PDFs for its Website, make print-ready files for the printer, and potentially create ebooks in the future. All formats will be generated from the same source, and all will be created using different style sheets to process the base XML files.

Well-Formedness and Validity

We've talked a little bit about XML, what it's used for, how it looks, how to conceptualize it, and how to transform it. One of the most powerful advantages of XML, of course, is that it allows you to define your own language.

However, this most powerful feature also exposes a great weakness of XML. If all of us start defining our own languages, we run the risk of being unable to understand anything anyone else says. Thus, the creators of XML had to set down some rules that would describe a "legal" XML document.

There are two levels of "legality" in XML:

A *well-formed* XML document follows these rules (most of which we've already discussed):

- An XML document must contain a single root element that contains all other elements.
- All elements must be properly nested.
- All elements must be closed either with a closing tag or with a "self-closing" empty-element tag (i.e. `<tag/>`).
- All attribute values must be quoted.

A *valid* XML document is both well-formed and follows all the rules set down in that document's DTD (document type definition). A valid document, then, is nothing more then a well-formed document that adheres to its DTD.

The question then becomes, why have two levels of legality? A good question, indeed!

For the most part, you will only care that your documents are well formed. In fact, most XML parsers (software that reads your XML documents) are non-validating (i.e. they don't care if your documents are valid) – and that includes those found in Web browsers like Firefox and Internet Explorer. Well-formedness alone allows you to create ad hoc XML documents that can be generated, added to an application, and tested quickly.

For other applications that are more mission-critical, you'll want to use a DTD within your XML documents, then run those documents through a validating parser.

The bottom line? Well-formedness is mandatory, but validity is an extra, optional step.

In the next section, we'll practice using both validating and non-validating parsers to get the hang of these tools.

Getting Your Hands Dirty

Okay, we've spent some time talking about XML and its potential, and examining some of the neater aspects of it. Now, it's time to do what I like best, and get our hands dirty as we actually work on some documents.

The first thing we want to do is to create an XML document. For our purposes, any XML document will do, but for the sake of continuity, let's use the product listing document we saw earlier in the chapter.

Here it is again, with a few more nodes added to it:

Example 1.1. `myFirstXML.xml`

```
<productListing title="ABC Products">
  <product>
    <name>Product One</name>
    <description>Product One is an exciting new widget that will
      simplify your life.</description>
    <cost>$19.95</cost>
    <shipping>$2.95</shipping>
  </product>
  <product>
```

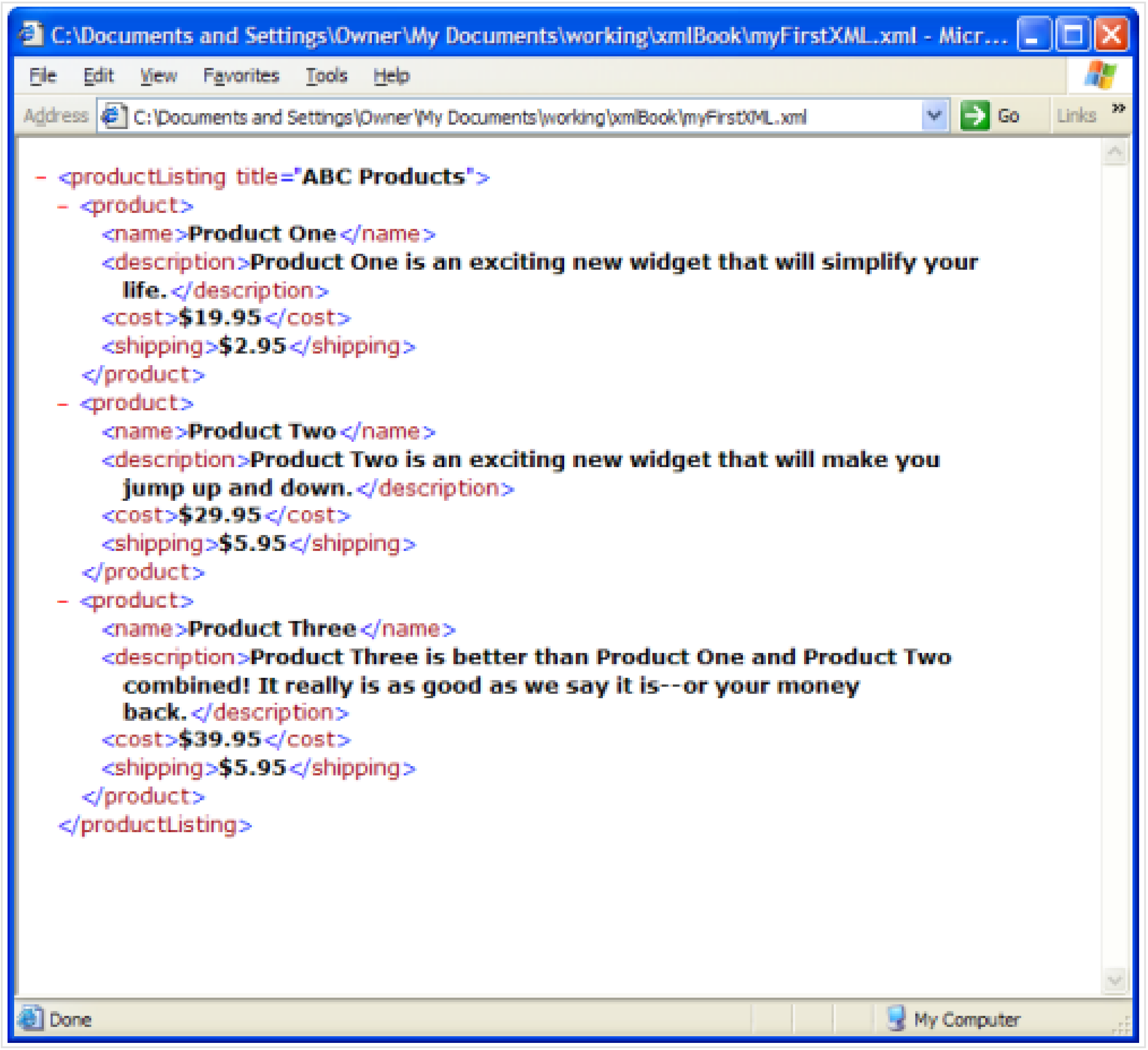
```
    <name>Product Two</name>
    <description>Product Two is an exciting new widget that will
      make you jump up and down.</description>
    <cost>$29.95</cost>
    <shipping>$5.95</shipping>
  </product>
  <product>
    <name>Product Three</name>
    <description>Product Three is better than Product One and
      Product Two combined! It really is as good as we say it
is--or your money back. </description>
    <cost>$39.95</cost>
    <shipping>$5.95</shipping>
  </product>
</productListing>
```

Save this XML markup into a file and name it `myFirstXML.xml` . In the next few sections, we'll be viewing the file in different browsers and experimenting with parsers.

Viewing Raw XML in Internet Explorer

If you have Internet Explorer 5 or higher installed on your machine, you can view your newly-created XML file. As Figure 1.2, "Viewing an XML file in Internet Explorer." illustrates, Internet Explorer simply displays XML files as a series of indented nodes.

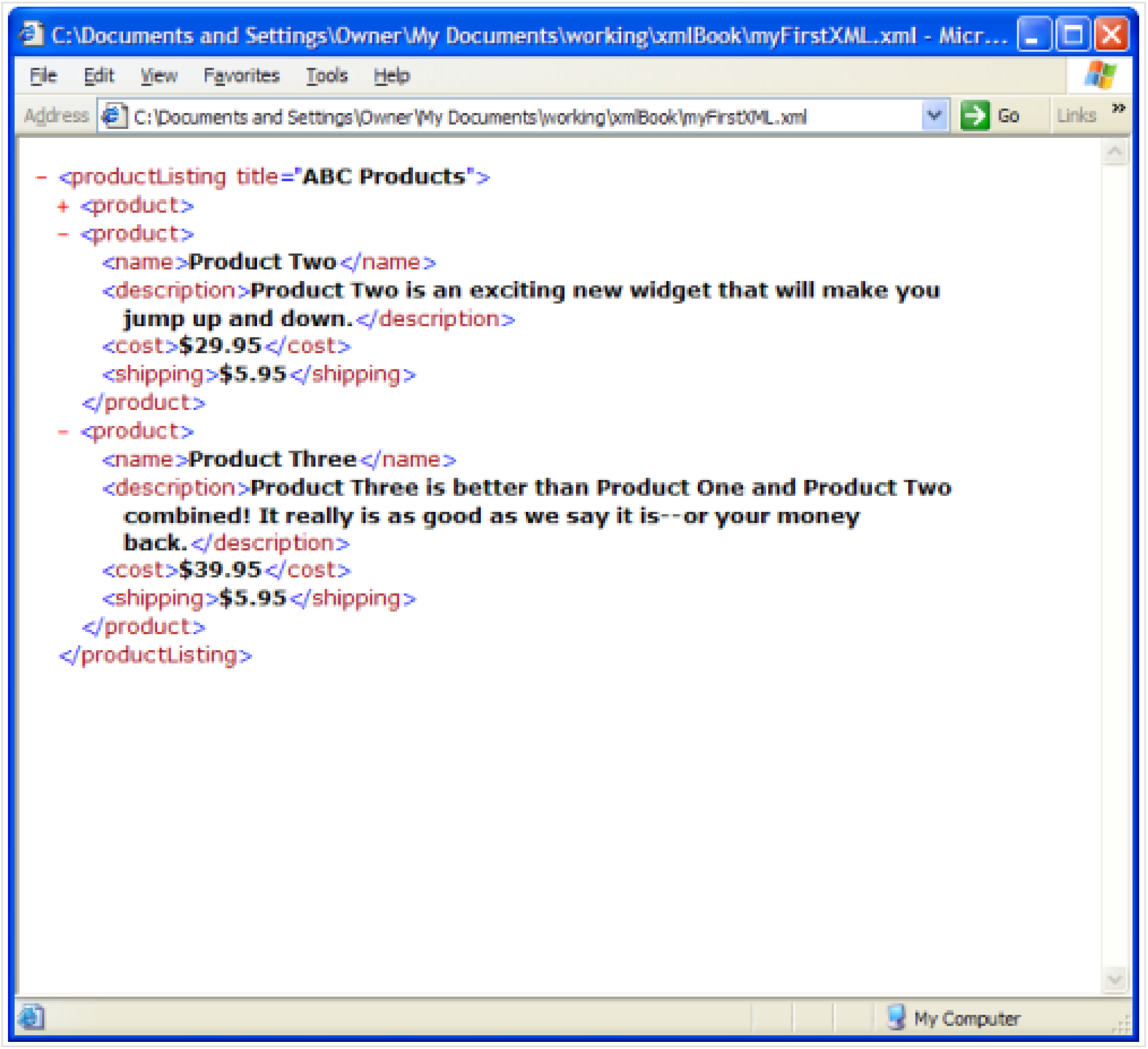
Figure 1.2. Viewing an XML file in Internet Explorer.


C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml - Micr...
File Edit View Favorites Tools Help
Address C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml
Go
Links
Done
My Computer

```
- <productListing title="ABC Products">
  - <product>
      <name>Product One</name>
      <description>Product One is an exciting new widget that will simplify your
        life.</description>
      <cost>$19.95</cost>
      <shipping>$2.95</shipping>
    </product>
  - <product>
      <name>Product Two</name>
      <description>Product Two is an exciting new widget that will make you
        jump up and down.</description>
      <cost>$29.95</cost>
      <shipping>$5.95</shipping>
    </product>
  - <product>
      <name>Product Three</name>
      <description>Product Three is better than Product One and Product Two
        combined! It really is as good as we say it is--or your money
        back.</description>
      <cost>$39.95</cost>
      <shipping>$5.95</shipping>
    </product>
  </productListing>
```

Notice the little minus signs next to some of the XML nodes? A minus sign in front of a node indicates that the node contains other nodes. If you click the minus sign, Internet Explorer will collapse all the child nodes belonging to that node, as shown in Figure 1.3, “Collapsing nodes displaying in Internet Explorer.”.

Figure 1.3. Collapsing nodes displaying in Internet Explorer.


C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml - Micr...
File Edit View Favorites Tools Help
Address C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml
Go
Links
My Computer

```
- <productListing title="ABC Products">
  + <product>
  - <product>
      <name>Product Two</name>
      <description>Product Two is an exciting new widget that will make you
        jump up and down.</description>
      <cost>$29.95</cost>
      <shipping>$5.95</shipping>
    </product>
  - <product>
      <name>Product Three</name>
      <description>Product Three is better than Product One and Product Two
        combined! It really is as good as we say it is--or your money
        back.</description>
      <cost>$39.95</cost>
      <shipping>$5.95</shipping>
    </product>
  </productListing>
```

View larger image.

The little plus sign next to the first product node indicates that the node has children. Clicking on the plus sign will expand any nodes under that particular node. In this way, you can easily display the parts of the document on which you want to focus.

Now, open your XML document in any text editing tool and scroll down to the cost node of the second product. The line we're interested in should read:

Example 1.2. `myFirstXML.xml` **(excerpt)**

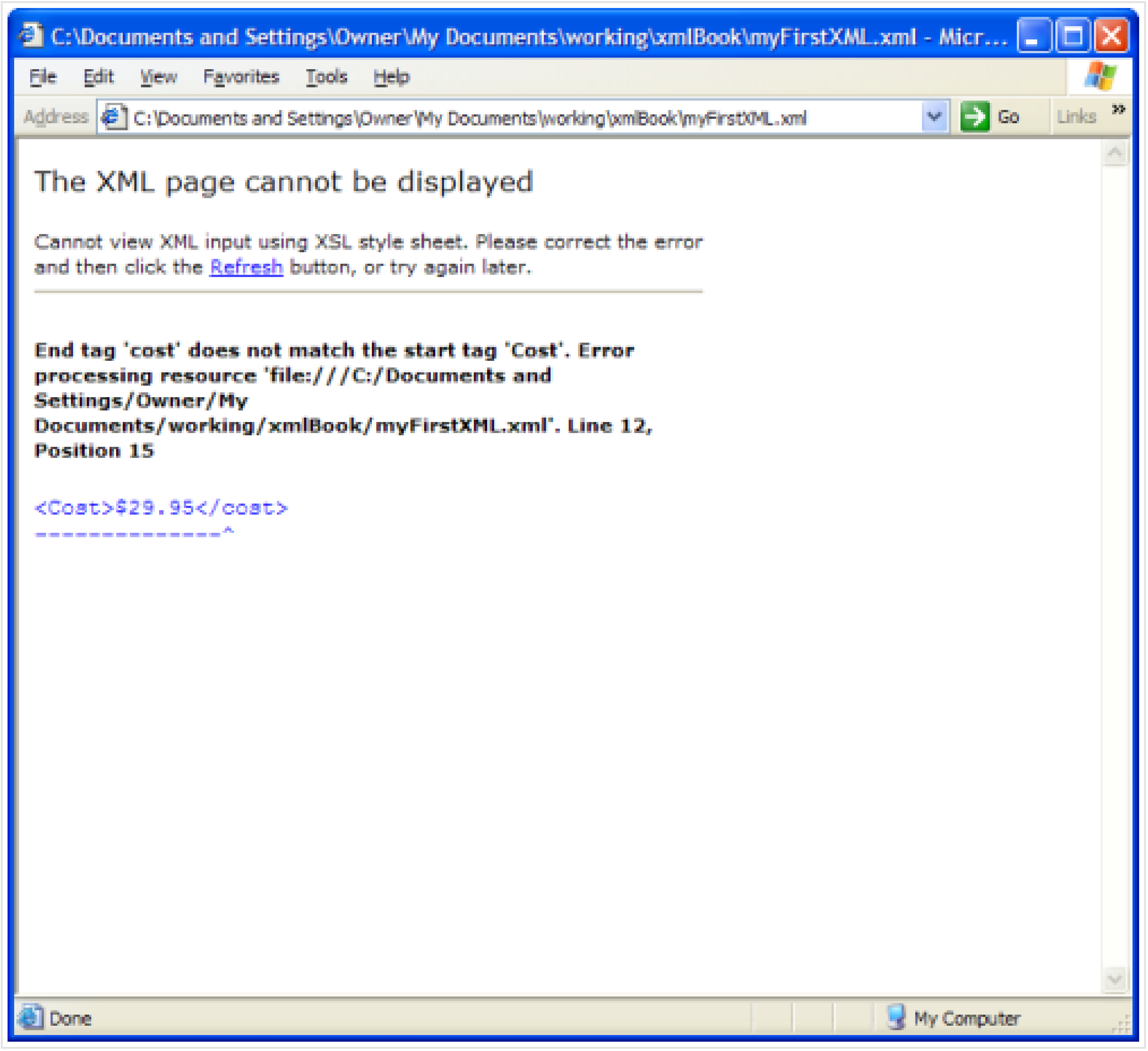
```
<cost>$29.95</cost>
```

Capitalize the " `c` " on the opening tag, so that the line reads like this:

```
<Cost>$29.95</cost>
```

Save your work and reload Internet Explorer. You should see an error message that looks like the one pictured in Figure 1.4, "Error message displaying in Internet Explorer.".

Figure 1.4. Error message displaying in Internet Explorer.


C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml - Micr...
File Edit View Favorites Tools Help
Address C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml
Go
Links
Done
My Computer

The XML page cannot be displayed

Cannot view XML input using XSL style sheet. Please correct the error and then click the Refresh button, or try again later.

End tag 'cost' does not match the start tag 'Cost'. Error processing resource 'file:///C:/Documents and Settings/Owner/My Documents/working/xmlBook/myFirstXML.xml'. Line 12, Position 15

```
<Cost>$29.95</cost>
--------------^
```

View larger image.

As you can see, Internet Explorer provides a rather verbose explanation of the error it ran into: the end tag, `</cost>` , does not match the start tag, `<Cost>` .

Furthermore, it provides a nice visual of the offending line, a little arrow pointing to the spot at which the parser thinks the problem arose.

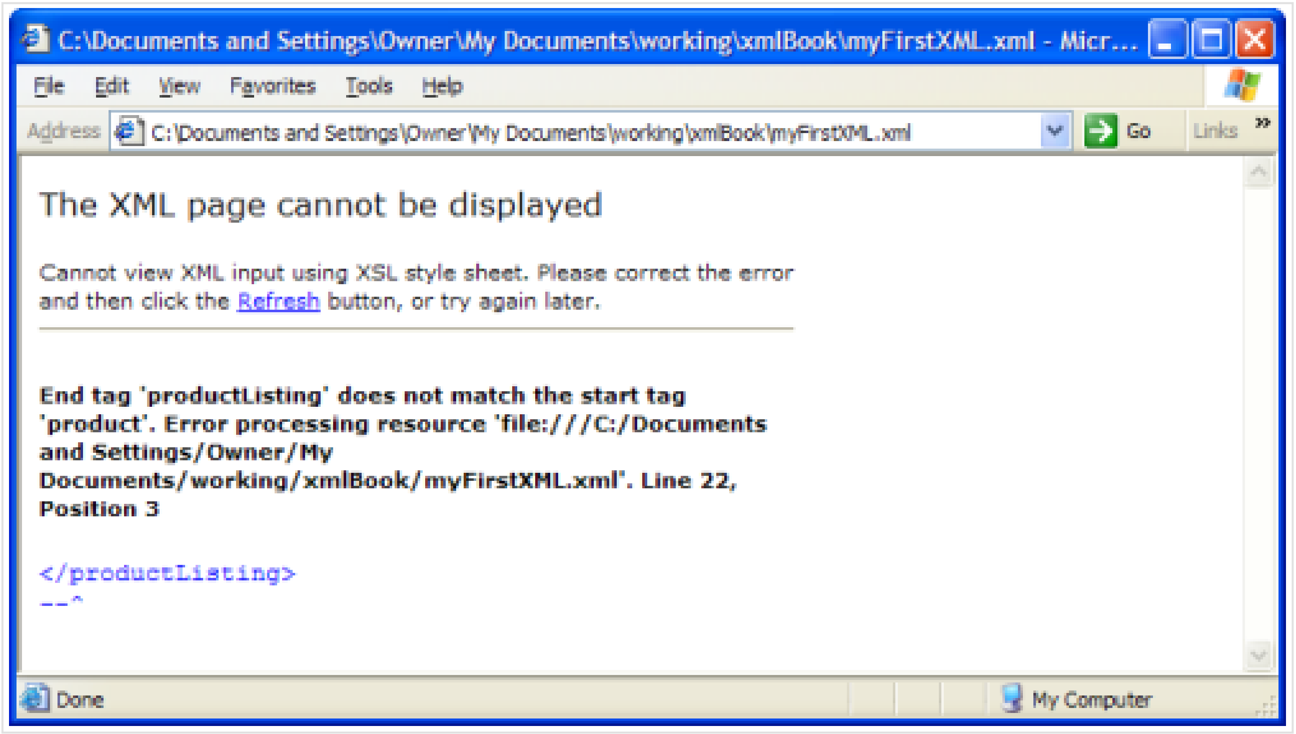
```
<Cost>$29.95</cost>
--------------^
```

Even though the problem is really with the start tag, the arrow points to the end tag. Because Internet Explorer uses a non-validating parser by default (remember, this means it only cares about well-formedness rules), it runs into problems at the end tag. You now have to backtrack to find out why that particular end tag caused such a problem. Once you get the hang of this debugging method, you'll find it a great help in tracking down problems.

Let's introduce a slightly more complex problem. Open your XML document in an editor once more, and fix the problem we introduced above. Then, go to the second-last line of the document (it should read `</product>`) and add a `<product>` tag in front of it. Save your work and reload your browser.

You should see an error message similar to the one shown in Figure 1.5, "Debugging a more complex error.".

Figure 1.5. Debugging a more complex error.


C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml - Micr...
File Edit View Favorites Tools Help
Address C:\Documents and Settings\Owner\My Documents\working\xmlBook\myFirstXML.xml
Go
Links
Done
My Computer

The XML page cannot be displayed

Cannot view XML input using XSL style sheet. Please correct the error and then click the Refresh button, or try again later.

End tag 'productListing' does not match the start tag 'product'. Error processing resource 'file:///C:/Documents and Settings/Owner/My Documents/working/xmlBook/myFirstXML.xml'. Line 22, Position 3

```
</productListing>
--^
```

View larger image.

At first glance, this error message seems a bit more obscure than the previous one. For starters, this message seems to indicate a problem with the `</productListing>` end tag. However, look closely and what do you see? It says that the `</productListing>` end tag does not match the `<product>` start tag. That's exactly what's wrong! Someone introduced a `<product>` start tag and didn't close it properly.

I'm including this example because bad nesting is one of the most common errors introduced to XML documents. This kind of error can be subtle and hard to find, especially if you're doing a lot of editing, or if your document is complex or long.

Viewing Raw XML in Firefox

You can also use Firefox (and other Mozilla browsers like Netscape 8) to view your XML files. Firefox is a popular open-source browser, and at the time this book went to print the latest version was 1.0.4. You can download a free copy from the Mozilla website.

Viewing raw XML in Firefox is basically the same as viewing it in Internet Explorer, as you can see from Figure 1.6, "Viewing raw XML in Firefox.".

Firefox's built-in parser is non-validating, so you won't be able to use it to check for document validity. However, it's comforting to know that the good folks at the Mozilla Foundation are planning to add a validating parser in a future release of the browser.

Options for Using a Validating Parser

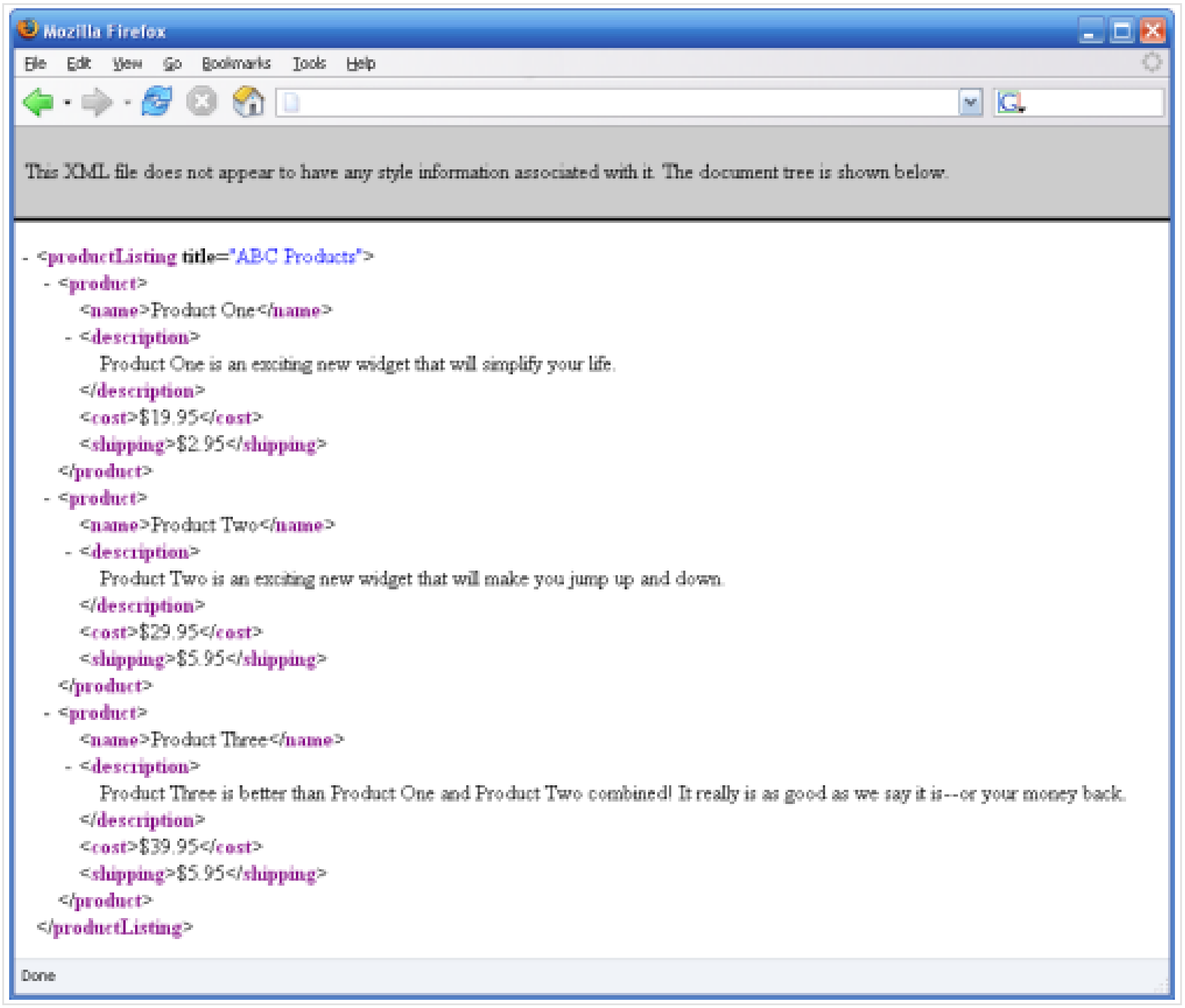
Okay, so both Internet Explorer and Firefox will check your XML for well-formedness, but you need to know for future reference how to check that an XML file is valid (i.e. conforms to a DTD). How do you do that?

Well, there are a couple of options, listed below.

Using an Online Validating Parser

There are various well-known online validating XML parsers. All you have to do is visit the appropriate page, upload your document, and the parser will validate it. Here is the most popular online parser.

Figure 1.6. Viewing raw XML in Firefox.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <productListing title="ABC Products">
  - <product>
      <name>Product One</name>
    - <description>
        Product One is an exciting new widget that will simplify your life.
      </description>
      <cost>$19.95</cost>
      <shipping>$2.95</shipping>
    </product>
  - <product>
      <name>Product Two</name>
    - <description>
        Product Two is an exciting new widget that will make you jump up and down.
      </description>
      <cost>$29.95</cost>
      <shipping>$5.95</shipping>
    </product>
  - <product>
      <name>Product Three</name>
    - <description>
        Product Three is better than Product One and Product Two combined! It really is as good as we say it is--or your money back.
      </description>
      <cost>$39.95</cost>
      <shipping>$5.95</shipping>
    </product>
  </productListing>
```

View larger image.

Using a Local Validating Parser

Sometimes, it may be impractical to use a Website to validate your XML because of issues relating to connectivity, privacy, or security. In any of these cases, it's a good idea to download one of the freely available solutions.

- If you're familiar with Perl, you can use any of the outstanding parser modules written for that language, all of which are available at CPAN.org.
- If you're comfortable with C++ or Visual Basic, then give MSXML by Microsoft a try.
- IBM offers a very good standalone validating parser called XML4J. Just download the package and install it by following the instructions provided. Be warned, however, that you will have to know something about working with

Java tools and files before you can get this one installed successfully.

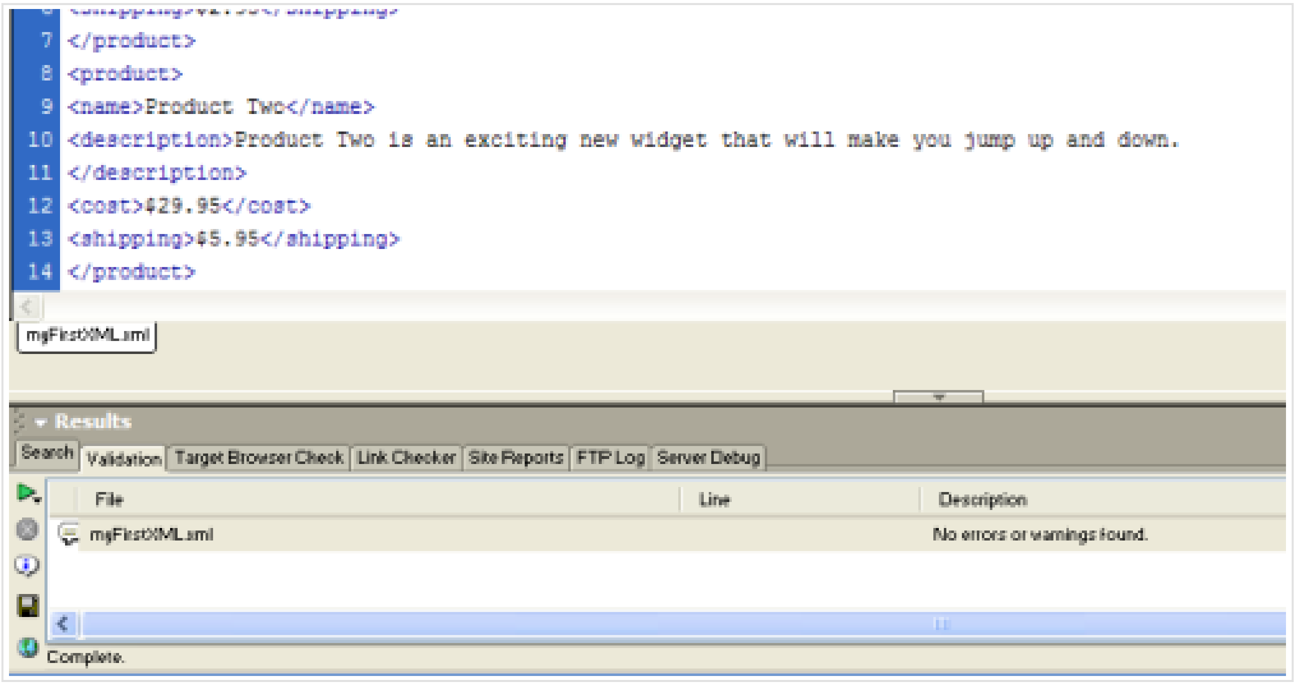
Using Dreamweaver

Dreamweaver isn't just a tool for creating Web pages; it's also an integrated development environment (IDE) that offers a suite of development tools to the interested Web developer.

One of Dreamweaver's more interesting capabilities is its built-in XML validator. This checks for well-formedness if the document has no DTD, and for well-formedness and validity if a DTD is specified. If you don't have a copy of Dreamweaver, you can get a trial version to play with.

To validate an XML document, choose File > Check Page in Dreamweaver, then select Validate as XML. Results of the validation will appear under the Results area, as illustrated in Figure 1.7, "Dreamweaver MX's validating XML parser."

Figure 1.7. Dreamweaver MX's validating XML parser.



```
</product>
<product>
<name>Product Two</name>
<description>Product Two is an exciting new widget that will make you jump up and down.
</description>
<cost>$29.95</cost>
<shipping>$5.95</shipping>
</product>
```

File	Line	Description
myFirstXML.xml		No errors or warnings found.

Complete.

View larger image.

What if I Can't Get a Validating Parser?

If you can't get your hands on a validating parser, don't panic. For most purposes, an online resource will do the job nicely. If you work in a company that has an established software development group, chances are that one of the XML-savvy developers has already set up a good validating parser.

What about the content management system we'll work on through the course of this book? Well, we won't need to validate our XML documents until we get close to the project's end, when we start to deal with Web Services, and need to figure out how to accept XML content from (and send content to) organizations in the world at large.

Starting Our CMS Project

Now that we've introduced XML and played around with some documents and parsers, it's time to start our project. Throughout this book, we'll spend time building an XML-powered Website. Specifically, we're going to build an XML-powered content management system. This project will help ground your skills as you obtain firsthand experience with practical XML development techniques, issues, and processes.

So... What's a Content Management System?

A content management system (henceforth referred to as a CMS) is a piece of server-side software that's used to create, publish, and maintain content easily and efficiently on a Website. It usually consists of the following components:

- A data back-end (comprising XML or database tables) that contains all your articles, news stories, images, and other content.
- A data display component – usually templates or other pages – onto which your articles, images, etc., are "painted" by the CMS for display to site visitors.
- A data administration component. This usually comprises easy-to-use HTML forms that allow site administrators to create, edit, publish, and delete articles in some kind of secure workflow. The data administration portion of a CMS is usually the most complicated, and this is the section on which you'll likely spend most of your development time.

Over the past decade, CMSs have been created using a range of different scripting languages including Perl/CGI, ASP, TCL, JSP, Python, and PHP. Each of these languages has its own pros and cons, but we'll use PHP with XML to build our CMS.

Requirements Gathering

Before you build any kind of CMS, first you must gather information that defines the basic requirements for the project.

The goal of the CMS is to make things easier for those who need to develop and run the site. And making things easier means having to do more homework beforehand! Although you may groan at the thought of this kind of exercise, a set of well-defined requirements can make the project run a lot more smoothly.

What kind of requirements do we need to gather? Essentially, requirements fall into three major categories:

- What kind of content will the CMS handle? How is each type of content broken down? (The more complete your understanding of this issue, the easier it'll be to create and manage your XML files.)
- Who will be visiting the site, and what behaviors do these users expect to find? (For example, will they want to browse a hierarchical list of articles, search for articles by keyword, see links to related articles, or all three?)
- What do the site administrators need to do? (For example, they may need to log in securely, create content, edit content, publish content, and delete content. If your CMS will provide different roles for administrative users – such as site administrators, editors, and writers – your system will become more complex.)

As you can see, we've barely scratched the surface, and already we've uncovered a number of issues that need addressing. Let's tackle them one at a time.

CMS Content and Metadata

If you're going to build a content management system, it's logical to expect that you're going to want to put content into it. However, it's not always that easy!

The most common failing I've seen on dozens of CMS engagements on which I've worked is that most of the companies that actually take the time to think about content only think about one thing: "articles!" I'm not exactly sure why that is, but I'd venture to guess that articles are what most folks are exposed to when they read newspapers, magazines, or Websites, so it's the first – and only – content type that comes to mind.

But if you're going to build a workable CMS, you'll have to think beyond "articles" and define your content types more clearly. There's a whole range of content types that need management: PDFs, images, news stories, multimedia presentations, user reviews of whitepapers/PDFs, and much, much more. In the world of XML, each of these different types of content is, naturally enough, called a *document type*.

The second most common failing I see is an inability to successfully convince site owners that content means more than just "articles." What's even harder is to convince them that you have to know as much as you possibly can about each content type if you're going to successfully build their CMS.

It's not good enough to know that you'll be serving PDF files, news stories, images, and so on. You also have to know how each of these content types will break out into its separate components, or *metadata*. Metadata means "data about data" and it is immensely useful to the CMS developer. Each article, for instance, will have various pieces of metadata, such as a headline, author name, and keywords, each of which the CMS needs to track.

The only way to understand a content type's metadata is to research it – in other words, ask yourself and others a whole lot of questions about that piece of content.

The final challenge – to define various types of metadata – can be a blessing in disguise. In my experience, once people grasp the importance of metadata, they race off in every direction and collect every single piece of metadata they can find about a given content type. Usually, we developers end up with random bits of information that aren't very useful and will never be used. For example, the client might start to track the date on which an article is first drafted. In most cases, this is unimportant information – the reader certainly doesn't care!

Obviously, it's important to look for the right kinds of metadata, like these:

Provenance Metadata

- Who created the content? When? When was it first published? When should it automatically be removed from the site, or archived? How is this document uniquely identified in the system? Who holds the copyright to it?

Organizational/Administrative Metadata

- If you're using category listings for your content, where will any individual piece of content live within that category system? What other content is it related to? Which keywords describe the content for indexing or search purposes (in other words, how do we find the content)? Who should have access to the content (the entire public, only site subscribers, or company staff)?

Physical/Structural Metadata

- Is the content ASCII text, an XML snippet, or a binary file, like a PDF or image? If it's a file, where does it reside on the server? What is the file's MIME type?

Descriptive Metadata

- If it's an article, what's the headline? Does the CMS view an article body as being separate from headings and paragraphs, or are all these items seen as one big lump of XML?

Gathering metadata can be very tricky. Let's take a look at a seemingly trivial issue: handling metadata about authors of articles. At first glance, we could say that all of our articles should contain elements for author name and email address, and leave it at that. However, we may later decide that we want site visitors to search or browse articles by author. In this case, it would make more sense to have a centralized list of authors, each with his or her own unique ID. This would eliminate the possibility of our having Tom Myer and Thomas Myer as "separate" authors just because the name was entered differently in individual articles.

Having a separate author listing would also allow us to easily set bylines for each author, in case someone decided they wanted to publish pieces under a pen name. It would also allow us to track author information across content types. We'd know, for instance, if a particular author has penned articles, written reviews, or uploaded files. Of course, agreeing on this approach means that we need to do other work later on, such as building administrative interfaces for author listings.

Once you've figured out the metadata required for a given content type, you can move on to the next content type. Eventually, you'll have a clear picture of all the content types you want your site to support.

What's the point of all this activity? Well, just think of metadata as one of the three pillars of your XML-powered CMS. (The other two are site functionality and site design. In many ways, metadata affect both and, thus, the user's experience of your site.) *Every piece of metadata could potentially drive some kind of site behavior, but each piece of metadata also must be managed by the administration tools you set up*.

Site Behavior

Site behavior should always be based on (and driven by) metadata. For example, if you're collecting keywords for all of your articles, you should be able to build a keyword-driven search engine for your site. If you're not collecting keyword information and want a keyword-driven search engine, you'd better back up and figure out how to add that to your content types.

Typical site behavior for a CMS-powered Website includes browsing by content categories, browsing by author, searching on titles and keywords, dynamic news sidebars, and more. Additionally, many XML- and database-powered sites feature homepages that boast dynamically updated content, such as Top Ten Downloads, latest news headlines, and so on.

CMS Administration

Our CMS will need to have an administrative component for each content type. It will also have to administer pieces of information that have nothing to do with content types, such as which users are authorized to log in to the CMS, and the privileges each of them has.

It goes without saying that your administrative interface has to be secure, otherwise, anyone could click to your CMS and start deleting content, making unauthorized changes to existing content, or adding new content that you may not want to have on your site.

In cases in which more than one person or department is involved with publishing content via the CMS, you'll need to consider workflow. A workflow is simply a set of rules that allow you to define who does what, when, and how. For example, your workflow might stipulate that a user with writer privileges may create an article, but that only a production editor can approve that content for publication on the site.

In many cases, CMS workflows emulate actual workflows that exist in publication and marketing departments. Because we're dealing with XML, we have a great opportunity to build a workflow system that's modular and flexible enough to take into account different requirements.

Defining your Content Types

We want to publish articles and news stories on our site. We definitely want to keep track of authors and site administrators, and we also want to build a search engine. We will also need to keep a record of all the copy on each of our site's pages, as well as binary files such as images and PDFs. That's a lot of work! For now, let's just step through the process of defining an article.

You may be asking, "Why are we messing around with content types at all?" It does seem like a silly thing for a developer to be doing, but it's actually the most vital task in building an XML-powered site. Whenever I build an XML-powered application, I try to define the content types first, because I find that all the other elements cascade from there. Because we've already spent some time discussing the structure of XML documents, and gathering requirements for the documents that will reside in our system, let's jump right in and start to define our article content type.

Articles

The articles in our CMS will be the mainstay of our site. In addition to the article text, each of our articles will be endowed with the following pieces of metadata:

- A unique identifier
- A headline
- A short description
- An author
- A keyword listing
- A publication date, which records when an article went live
- Its status

Our article content type requires a root element that contains all the others; we can use `<article>` as that element. This not only makes sense from a "keep it simple" standpoint, but it is semantically appropriate, too.

Furthermore, because we need to identify each article in our system uniquely with an ID of some sort, it makes sense to add an id attribute to the root element that will contain this value. A unique identifier will ensure that no mistakes occur when we try to edit, delete, or view an existing article.

Now, each of our articles will have an author, so we need to reserve a spot for that information. There are literally dozens of ways to do this, but we'll take the simplest approach for now:

```
<article id="123">
  <author>Tom Myer</author>
 </article>
```

Looking for the DTD?

In Chapter 3, *DTDs for Consistency*, we'll discuss document type definitions (DTDs) – the traditional means to structure the rules for an XML file – in detail. For now, I think it makes more sense to continue our discussion in the direction we've already chosen.

Our article will need a headline, a short description, a publication date, and some keywords. The `<headline>` is very simple – it can have its own element nested under the `<article>` element. Likewise, the `<description>` and `<pubdate>` elements will be nested under `<article>` .

The keyword listing can be handled in one of two ways. You could create under `<article>` a /c#/<keywords> element that itself was able to contain numerous `<keyword>` items:

```
<article id="123">
  <author>Tom Myer</author>
  <headline>Creating an XML-powered CMS</headline>
  <description>This article will show you how to create an
    XML-powered content management system</description>
  <pubdate>2004-01-20</pubdate>
  <keywords>
    <keyword>XML</keyword>
    <keyword>CMS</keyword>
  </keywords>
 </article>
```

This approach will satisfy the structure nuts out there, but it turns out to be too complicated for the way we will eventually use these keywords. It turns out that all you really need is to list your keywords in a single `<keywords>` element, separated by spaces:

```
<article id="123">
```

```
  <author>Tom Myer</author>
  <headline>Creating an XML-powered CMS</headline>
  <description>This article will show you how to create an
    XML-powered content management system</description>
  <pubdate>2004-01-20</pubdate>
  <keywords>XML CMS</keywords>
</article>
```

Since individual keywords won't really have any importance in our system, this way of storing them works just fine.

Let's take a look at our growing XML document:

```
<article id="123">
  <author>Tom Myer</author>
  <headline>Creating an XML-powered CMS</headline>
  <description>This article will show you how to create an
    XML-powered content management system</description>
  <pubdate>2004-01-20</pubdate>
  <keywords>XML CMS</keywords>
</article>
```

We also need to track status information on the article. Because we don't need very robust workflows in this application, we can keep our status list very short, to "in progress" and "live."

Any article that is "in progress" will not be displayed on the live Website. It's a piece of content that's being worked on internally. Any article that is "live" will be displayed.

The easiest way to keep track of this information is to add a `<status>` element to our document:

```
<status>in progress</status>
```

However, you probably already see that status is very similar to keyword listings in that it has the potential to belong to many different content types. As such, it makes sense to centralize this information. We'll address this issue later, but for now, we'll continue to store status information in each article.

Now, we have to do something about the article's body. As most of our content will be displayed in a Web browser, it makes sense to use as many tags as possible that a browser like IE or Firefox can already understand. So HTML will form the basis of our article body's code. But for the purposes of our article storage system, we want to treat all of the HTML tags and text that make up the document body as a simple text string, rather than having to handle every single HTML tag that could appear in the article body. The best way to do this is to use a *CDATA section* within our XML document. XML parsers ignore tags, comments, and other XML syntax within a CDATA section – it simply passes the code through as a text string, without trying to interpret it. Here's what this looks like:

```
 <body>[CDATA[
```

<h1>Creating an XML-powered CMS</h1>

<p>Here is all of our paragraph information. . .</p>

]]</body>

Well, we're done with articles! They now look like this:

```
 <article id="123">
   <author>Tom Myer</author>
   <headline>Creating an XML-powered CMS</headline>
   <description>This article will show you how to create an
     XML-powered content management system</description>
   <pubdate>2004-01-20</pubdate>
   <status>live</status>
   <keywords>XML CMS</keywords>
   <body>[CDATA[
     <h1>Creating an XML-powered CMS</h1>
     <p>In this article...</p>
   ]]</body>
 </article>
```

Gathering Requirements for Content Display

We now understand our article content type, which defines most of the content we'll display on the site. Now, let's talk about our requirements for displaying content.

- The display side of our site will only display articles and other content that

have a status of “live.”
- The search engine will retrieve content by keywords, titles, and descriptions, and only display those pieces that have a status of live.
- The Website will display a list of author names by which site visitors can browse content, but it will only display those authors who have live articles posted on the site.

Gathering Requirements for the Administrative Tool

Let’s talk briefly about the administrative tool. Here are some of the project’s administration requirements:

- All CMS users must log into the administrative tool. All passwords set for administrators will be encrypted before they’re stored.
- Each content type will have its own page through which users may list, add, edit, and delete individual pieces of content.
- The same is true for authors and administrators. If you view an author listing, the CMS will display all pieces of content authored by that person.
- The CMS will provide an easy method to update status, keyword, and other details for each piece of content on the site. Administrators will be able to group this information by status or content type.

Great – this is enough detail to get us started!

Summary

In this first chapter, we’ve discussed basic XML concepts, talked about the importance of the requirements gathering process, and performed an analysis to come up with content types and application requirements for our XML-powered CMS.

In the next chapter, we’re going to delve deeper into XML, covering such topics as basic XSLT and XPath. We’ll get our hands dirty with a little XSLT and start thinking about how we should display articles on our CMS-powered Website.

Chapter 2. XML in Practice

The last chapter introduced some basic concepts in XML and saw us start our CMS project. In this chapter, we’re going to dig a little deeper into XML as we talk about namespaces, XHTML, XSLT, and CSS. In the process, we’ll have take a couple of opportunities to make XML *do* something.

Meet the Family

In Chapter 1, *Introduction to XML*, we learned a few things about how XML is structured and what you can do with it. My goal for that chapter was to show you how flexible XML really is.

In this chapter, I'd like to zoom out a little and introduce you to some of the wacky siblings that make up the XML "Family of Technologies." Although I'm going to list a number of tools and technologies here, we'll cover only a few in this chapter. We'll explore some of the others in later chapters, but some will not be covered at all (sorry, but this would be a very long and boring book if we gave equal space to everything).

XSLT

XSLT stands for Extensible Stylesheet Language Transformations. It is both a style sheet specification and a kind of programming language that allows you to transform an XML document into the format of your choice: stripped ASCII text, HTML, RTF, and even other dialects of XML. In this chapter, you'll be introduced to XSLT concepts; later in the book, we'll explore these in more depth. XSLT uses XPath and several other technologies to do its work.

XPath

XPath is a language for locating and processing nodes in an XML document. Because each XML document is, by definition, a hierarchical structure, it becomes possible to navigate this structure in a logical, formal way (i.e. by following a path).

DTD and XML Schema

A document type definition (DTD) is a set of rules that governs the order in which your elements can be used, and the kind of information each can contain. XML Schema is a newer standard with capabilities that extend far beyond those of DTDs. While a DTD can provide only general control over element ordering and containment, schemas are a lot more specific. They can, for example, allow elements to appear only a certain number of times, or require that elements contain specific types of data such as dates and numbers.

Both technologies allow you to set rules for the contents of your XML documents. If you need to share your XML documents with another group, or you must rely on receiving well-formed XML from someone else, these technologies can help ensure that your particular set of rules is properly followed. We will explore both of these technologies with loving attention in Chapter 3, *DTDs for Consistency*.

XML Namespaces

The ability of XML to allow you to define your own elements provides flexibility and scope. But it also creates the strong possibility that, when combining XML content from different sources, you'll experience clashes between code in which the same element names serve very different purposes. For example, if you're running a bookstore, your use of `<title>` tags in XML may be used to track book titles. A mortgage broker would use `<title>` in a different way – perhaps to track the title on a deed. A dentist or doctor might use `<title>` to track patients' formal titles (Mr., Ms., Mrs., or Dr.) on their medical records. Try to combine all three types of information into one system (or even one document), and you'll quickly see how problems can arise.

XML namespaces attempt to keep different semantic usages of the same XML elements separate and unambiguous. In our example, each person could define their own namespace and then prepend the name of their namespace to specific tags: `<book:title>` is different from `<broker:title>` and `<medrec:title>`. Namespaces, by the way, are one of the technologies that make XSLT and XSD work.

XHTML

XHTML stands for Extensible Hypertext Markup Language. Technically speaking, it's a reformulation of HTML 4.01 as an application of XML, and is not part of the XML family of technologies. To save your brain from complete meltdown, it might be simplest to think of XHTML as a standard for HTML markup tags that follow all the well-formedness rules of XML we covered earlier.

What's the point of that, you might ask? Well, there are tons and tons and *tons* of Websites out there that already use HTML. No one in their right mind could reasonably expect them all to switch to XML overnight. But we *can* expect that some of these pages – and a large percentage of the new pages that are being coded as you read this – will make the transition thanks to XHTML.

As you can see, the XML family of technologies is a pretty big group – those XML family reunions are undoubtedly interesting! It's also important to note that these technologies are open standards-based, which means that any new XML technologies (or proposed changes to existing ones) must follow a public process set down by the W3C (the World Wide Web Consortium) in order to gain acceptance in the community.

Although this means that some ideas take quite a while to reach fruition, and tend to be built by committee, it also means that no single vendor is in total control of XML. And this, as Martha Stewart might say, is a good thing.

A Closer Look at XHTML

Imagine you're at a cocktail party and somebody asks, "Okay, what's XHTML *really*?" You needed to tell them something (besides, "Hey, I'm trying to have a relaxing cocktail here!"). So, what do you say? Not sure? That's what I thought.

Because this is a book about XML and not XHTML, and because there are plenty of terrific books out there on XHTML, I don't want to get into too much detail about the technology here. However, I do feel that a basic knowledge of XHTML will serve you well, and will help to reinforce the concepts we've already introduced.

So, back to our cocktail party. Here are some answers that you might give in that situation:

- XHTML stands for Extensible HyperText Markup Language.
- XHTML is designed to replace HTML.
- XHTML uses the HTML 4.01 tag set, but is written using the XML syntax rules.
- XHTML is a stricter, cleaner version of HTML.

Why do we need XHTML? Well, put bluntly, the Web has reached a point at which just about anything will fly when it comes to HTML documents. Take a look at the following snippet:

```
<html><title>My example</title>
 <h1>Hello</h1>
```

Believe it or not, that snippet will render without a problem in most Web browsers. And so will this:

```
<p><b><i>Hello</b>
```

So will this:

```
Hello
```

I don't want to start some kind of crusade about HTML structure, but hey, enough is enough! Web pages represent structured information, so please, let's at least maintain some semblance of structure! At its most basic, XHTML was designed to form a kind of bridge between the loosy-goosy world of HTML and the more rigid structure of XML.

Remember that list of statements about XHTML we saw a moment ago? Well, here's another way to think about XHTML:

XHTML consists of all HTML 4.01 elements combined with the syntax of XML.

Simple! But, exactly what does this mean? Well, if you recall what we said in Chapter 1, *Introduction to XML* about well-formed XML documents, you can make some very good guesses:

1. XHTML documents must contain a root element that contains all other elements. (In most cases, the `html` element!)
2. XHTML elements must be properly nested.

```
<p>This is a <b>sentence.</b></p>
```

- All XHTML elements must have closing tags (even empty ones).

```
<br />
 <td></td>
```

Don't Slash Backwards Compatibility

Older browsers, such as Netscape 4, which do not recognize XML syntax, will become confused by self-closing tags like `<br/>` . By simply adding a space before the slash `(<br />)` , you can ensure that these browsers will ignore the slash and interpret the tag correctly.

1. All XHTML attribute values must be placed between quotes.
   `<input type="button" name="submit" value="click to finish" />`

- All XHTML element and attribute names must be written in lowercase.
  `<tr valign="top">`

- Each XHTML document must have a DOCTYPE declaration at the top.
  `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"`

```
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

    <html xmlns="http://www.w3.org/1999/xhtml">
```

There are three XHTML DOCTYPES:

Strict

Use this with CSS to minimize presentational clutter. In fact, the Strict DOCTYPE expressly prohibits the use of HTML's presentation tags.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Transitional

Use this to take advantage of HTML's presentational features and/or when you're supporting non-CSS browsers.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Frameset

Use this when you want to use frames to partition the screen.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

A Minimalist XHTML Example

Here's a very simple document that illustrates the rules above:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>A very simple XHTML document</title>
<meta http-equiv="content-type"
    content="text/html; charset=iso-8859-1" />
```

```
</head>
<body>
<p>a simple paragraph that contains a properly formatted<br />
break and some <b><i>properly nested</i></b> formatting.</p>
<div><img src="myphoto.jpg" alt="notice that all my quotes are in
place for attribute values" /></div>
</body>
</html>
```

That's more than enough information about XHTML for the moment. Let's move on to discuss namespaces and XSLT.

XML Namespaces

XML Namespaces were invented to rectify a common problem: the collision of documents using identical element names for different data.

Let's revisit our namespace example from this chapter's introduction. Imagine you were running a bookstore and had an inventory file (called `inventory.xml`, naturally), in which you used a `title` element to store book titles. Let's also say that – unlikely though it sounds – your XML document becomes mixed in with a mortgage broker's master record file. In this file, the mortgage broker has used `title` to store information about a property's legal title.

A human being could probably figure out that one `title` has nothing to do with the other, but an application that tried to sort it out would go nuts. We need to have a way to distinguish between the two different semantic universes in which these identical terms exist.

Let's get even more ambiguous: imagine you had an `inventory.xml` file in your bookstore that used the `title` element to store book titles, and a separate `sales.xml` file that used the `title` element to store the same information, but in a completely different context. Your inventory file stores information about books on the shelf, but the sales file stores information about books that have been bought by customers.

In either situation, regardless of the chasm that lies between the contexts of these identical terms, we need a way to properly label each context.

Namespaces to the rescue! XML namespaces allow you to create a unique namespace based on a URI (Uniform Resource Identifier), give that namespace a prefix, and apply that prefix to XML document elements.

Declaring Namespaces

To use and declare a namespace, we must first tie the namespace to a URI. Notice that I didn't say URL – a specific location that you can reach (although a URI can be a URL). A URI is simply a unique identifier that distinguishes one thing (say, an XML document standard) from another. URIs can take the following forms:

URL

Uniform Resource Locator: a specific protocol, machine address, and file path (e.g. `http://www.tripledogdaremedia.com/index.php`).

URN

Uniform Resource Name: a persistent name that doesn't point to an actual location for the resource, but still identifies it uniquely. For example, all published books have an ISBN. The ISBN uniquely identifies the book, but nowhere in the ISBN is there any indication as to which shelf it sits on in any particular bookstore. However, armed with the ISBN, you could walk into the store, ask an employee to search for you, and they could take you right to the book (provided, of course, that it was in stock.

The following are examples of good URIs:

```
http://www.tripledogdaremedia.com/XML/Namespaces/1
 urn:bookstore-inventory-namespace
```

We want to use our namespace throughout our XML documents, though, and the last thing we want to do is type out an entire URI every time we need to distinguish one context from another. So, we define a prefix to represent our namespace to ease the strain on our typing fingers:

```
inv="urn:bookstore-inventory-namespace"
```

But, wait – we're not done yet! We need a way to tell the XML parser that we're creating a namespace. The agreed way to do that is to prefix the namespace declaration with `xmlns`:, like this:

```
xmlns:inv="urn:bookstore-inventory-namespace"
```

At this point, we have something useful. If we needed to, we could add our prefix to appropriate elements to disambiguate (I love that term!) any potentially ambiguous usage, like this:

```
<inv:title>Build Your Own XML-Powered Web Site</inv:title>
 <title>Title Deed to the house on 123 Main St., YourTown</title>
```

Namespaces make it very clear that `<inv:title>` is very different from `<title>`.

But, where do we put our namespace declaration?

Placing Namespace Declarations in your XML Documents

In most cases, placing your namespace declarations will be rather easy. They're commonly located in the root element of a document, like so:

```
<inventory xmlns:inv="urn:bookstore-inventory-namespace">
 ...
 </inventory>
```

Please note, however, that namespaces have scope. Namespaces affect the element in which they are declared, as well as all the child elements of that element. In fact, as you'll see when we discuss XSLT later, we'll use the `xsl` prefix in the very element in which we define the XSL namespace:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml"
    version="1.0">
```

Any namespace declaration that's placed in a document's root element becomes available to all elements in that document. However, if you want to limit your namespace scope to a certain part of a document, feel free to do so – remembering, of course, that this can get pretty tricky. My advice is to declare your namespaces in the document's root element, then use the prefixes when you need them.

Using Default Namespaces

It would become pretty tiresome to have to type a prefix for every single element in a document. Fortunately, you can declare a default namespace that doesn't contain a prefix. This namespace will apply to all elements that don't contain prefixes.

Let’s take another look at a typical opening `<xsl:stylesheet>` tag for an XSLT file:

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml"
    version="1.0">
```

Notice the non-prefixed namespace: `xmlns="http://www.w3.org/1999/xhtml"` In an XSLT file, this namespace governs all elements that aren’t specifically prefixed as XSLT elements, identifying them as XHTML tags. On the other side of the coin, all XSLT elements must be given the `xsl:` prefix.

Using CSS to Display XML In a Browser

The most powerful tools available for displaying XML in a browser are XSLT and Cascading Style Sheets (CSS). Because XSLT can be quite a tricky undertaking for newbies, I’ve decided to let you practice with CSS first!

The first step in working with CSS is to create a basic XML file:

Example 2.1. `letter.xml` (excerpt)

```
<?xml version="1.0"?>
 <letter>
  <to>Mom</to>
  <from>Tom</from>
  <message>Happy Mother's Day</message>
 </letter>
```

As XML documents go, this one could be made a lot simpler, but there’s no point in making things too simple. This document contains a root element (`letter`) that contains three other elements (`to` , `from` , and `message`), each of which contains text.

Now, we need to add a style sheet declaration that will point to the CSS document we’ll create. To associate a CSS style sheet with an XML file, use the `<?xml-stylesheet?>` directive:

Example 2.2. `letter-css.xml` (excerpt)

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

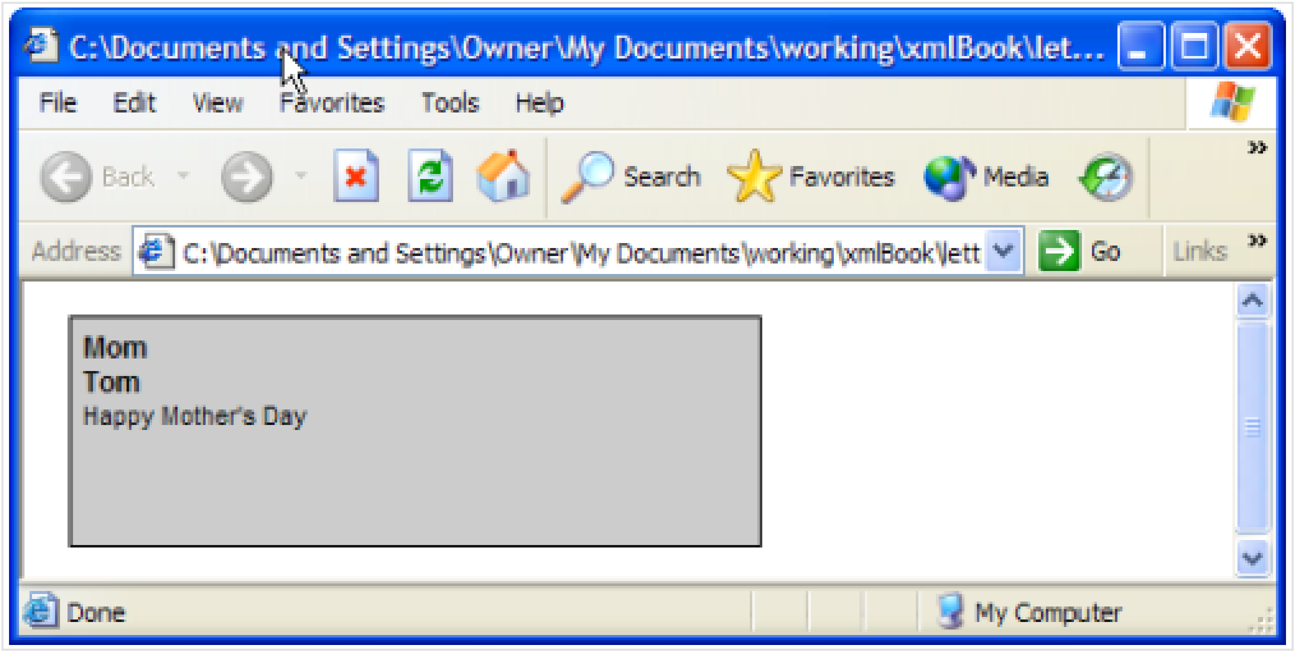
Finally, we write our CSS file, making sure that we provide a style for each element in our XML file:

Example 2.3. `letter.css`

```
letter {
  display: block;
  margin: 10px;
  padding: 5px;
  width: 300px;
  height: 100px;
  border: 1px solid #00000;
  overflow: auto;
  background-color: #cccccc;
  font: 12px Arial;
 }
 to, from {
  display: block;
  font-weight: bold;
 }
 message {
  display: block;
  font: 11px Arial;
 }
```

When you display your XML document, you should see something similar to Figure 2.1, "Viewing the CSS results in Internet Explorer.".

Figure 2.1. Viewing the CSS results in Internet Explorer.


C:\Documents and Settings\Owner\My Documents\working\xmlBook\let...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Documents and Settings\Owner\My Documents\working\xmlBook\lett Go Links
Done My Computer

Mom
Tom
Happy Mother's Day

View larger image.

As you can see, CSS did a marvelous job of rendering a nicely shaded box around the entire letter, setting fonts, and even displaying things like margins and padding. What it didn't allow us to do, however, was add text to the output. For instance, we could use a "To:" in front of whatever text was in the to element. If you want to have that kind of power, you'll need to use XSLT. Strictly speaking, the CSS standard does allow for this sort of thing with the content property, which can produce *generated text* before and after document elements. Many browsers do not support this property, however, and even those that do don't provide anywhere near the flexibility of XSLT.

Getting to Know XSLT

XSLT, as I mentioned earlier in the chapter, stands for Extensible Stylesheet Language Transformations. Think of it as a tool that you can use to transform your XML documents into other documents. Here are some of the possibilities:

- Transform XML into HTML or raw ASCII text.
- Transform XML into other dialects of XML.
- Pull out all the passages tagged as Spanish, or French, or German to create foreign-language versions of your XML document.

Not bad – and we've barely scratched the surface!

XSLT is a rules-based, or functional language. It's not like other programming languages (e.g. PHP or JSP) that are procedural or object-oriented. Instead, XSLT requires that you supply a series of rules (called "templates") that tell it what to do when it encounters the various elements of an XML document.

For instance, upon identifying an XML `<para>` tag in the input document, a rule could instruct XSLT to convert it into an HTML `<p>` tag.

Because XSLT can be a little bewildering even for veteran programmers, the best way to tackle it is to walk through a series of examples. That way, I can give you the practical information you'll need to get started, and you can learn the key concepts along the way. As with XHTML, countless books, articles, and Websites are devoted to XSLT; use these to continue your education.

Your First XSLT Exercise

Let's get started with XSLT. For our first exercise, we'll reuse the very simple Letter to Mother example we saw in the CSS section. We'll also create a very basic Extensible Stylesheet Language (XSL) file to transform that XML. Keeping both these elements simple will give us the opportunity to step through the major concepts involved.

First, let's create the XSL file. This file will contain all the instructions we'll need in order to transform the XML elements into raw text.

In what will become a recurring theme in the world of XML, XSL files are in fact XML files in their own right. They must therefore follow the rules that apply to all XML documents: an XSL file must contain a root element, all attribute values must be quoted, and so on.

All XSL documents begin with a `stylesheet` element This element contains information that the XSLT processor needs to do its job:

Example 2.4. `letter2text.xsl` **(excerpt)**

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

The `version` attribute is required. In most cases, you'd use `1.0` , as this is the most widely supported version at the time of this writing.

The `xmlns:xsl` attribute is used to declare an XML namespace with the prefix `xsl`. For your stylesheet transformation to work at all, you must declare an XML namespace for the URI `http://www.w3.org/1999/XSL/Transform` in your opening `<stylesheet>` tag. In our example, we will use an `xsl` prefix on all the stylesheet-related tags in our XSL documents to associate them with this namespace. You'll find this is common practice when working with XSLT.

The next element will be the `output` element, which is used to define the type of output you want from the XSL file. For this first example, we'll use `text` as our `method`:

Example 2.5. `letter2text.xsl` **(excerpt)**

```
<xsl:output method="text"/>
```

Other possible values for the `method` attribute include `html` and `xml`, but we'll cover those a little later.

Now we come to the heart of XSLT – the `template` and `apply-templates` elements. Together, these two elements make the transformations happen.

Put simply, the XSLT processor (for our immediate purposes, the browser) starts reading the input document, looking for elements that match any of the `template` elements in our style sheet. When one is found, the contents of the corresponding `template` element tells the processor what to output before continuing its search. Where a template contains an `apply-templates` element, the XSLT processor will search for XML elements contained *within* the current element and apply templates associated with them.

There are some exceptions and additional complications that we'll see as we move forward, but for now, that's really all there is to it.

The first thing we want to do is match the `letter` element that contains the rest of our document. This is fairly straightforward:

Example 2.6. `letter2text.xsl` **(excerpt)**

```
<xsl:template match="/letter">
```

<xsl:apply-templates select="*"/>

</xsl:template>

This very simple batch of XSLT simply states: "when you encounter a `letter` element at the root of the document, apply any templates associated with the elements it contains." Let's break this down.

The `<xsl:template>` tag is used to create a template, with the `match` attribute indicating which element(s) it should match. The value of this attribute is an XPath expression (we'll learn more about XPath later). In this case, the `/letter` value indicates that the template should match the `letter` elements at the root of the document. Were the value simply `letter`, the template would match `letter` elements throughout the document.

Now, this `<xsl:template>` tag contains only an `<xsl:apply-templates>` tag, which means that it doesn't actually output anything itself. Rather, the `<xsl:apply-templates>` tag sends the processor looking for other elements with matching templates.

By default, `apply-templates` will match not only elements, but text and even whitespace between the elements as well. XSLT processors have a set of default, or *implicit* templates, one of which simply outputs any text or whitespace it encounters. Since we want to ignore any text or whitespace that appears between the tags inside `<letter>`, we use the `select` attribute of `apply-templates` to tell the processor to look for child elements only in its search. We do this with another XPath expression: * means "all child elements of the current element."

Now, we've got our processor looking for elements inside `letter`, so we'd better give it some templates to match them!

Example 2.7. `letter2text.xsl` **(excerpt)**

`<xsl:template match="to">`

TO: <xsl:apply-templates/>

</xsl:template>

<xsl:template match="from">

FROM: <xsl:apply-templates/>

</xsl:template>

<xsl:template match="message">

MESSAGE: <xsl:apply-templates/>

</xsl:template>

Each of these templates matches one of the elements we expect to find inside the `letter` element: `to` , `from` , and `message` . In each case, we output a text label (e.g. TO:) and then use `apply-templates` to output the contents of the tag (remember, in the absence of a `select` attribute that says otherwise,apply-templates will output any text contained in the tags automatically).

The last thing we have to do in the XSL file is close off the `stylesheet` element that began the file:

```
</xsl:stylesheet>
```

Our style sheet now looks like this:

Example 2.8. `letter2text.xsl` **(excerpt)**

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/letter">
    <xsl:apply-templates select="*"/>
  </xsl:template>
  <xsl:template match="to">
    TO: <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="from">
    FROM: <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="message">
    MESSAGE: <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

While the logic of this style sheet is complete and correct, there's a slight formatting issue left to be tackled. Left this way, the output would look something like this:

```
 TO: Mom
  FROM: Tom
  MESSAGE: Happy Mother's Day
```

There's an extraneous line break at the top of the file, and each of the lines begins with some unwanted whitespace. The line break and whitespace is actually coming from the way we've formatted the code in the style sheet. Each of our three main templates begins with a line break and then some whitespace before the label, which is being carried through to the output.

But wait – what about the line break and whitespace that *ends* each template? Why isn't *that* getting carried through to the output? Well by default, the XSLT standard mandates that whenever there in *only* whitespace (including line breaks) between two tags, the whitespace should be ignored. But when there is text between two tags (e.g. `TO:`), then the whitespace in and around that text should be passed along to the output.

Avoid Whitespace Insanity

The vast majority of XML books and tutorials out there completely ignore these whitespace treatment issues. And while it's true that whitespace doesn't matter a lot of the time when you're dealing exclusively with XML documents (as opposed to formatted text output), it's likely to sneak up on you and bite you in the butt eventually. Best to get a good grasp of it now, rather than waiting for insanity to set in when you least expect it.

The `<xsl:text>` tag is useful for controlling the effects of whitespace in our style sheets. All it does is output the text it contains, even if it is just whitespace. Here's the adjusted version of our style sheet, with `<xsl:text>` tags used to isolate text we want to output:

Example 2.9. `letter2text.xsl`

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/letter">
    <xsl:apply-templates select="*"/>
  </xsl:template>
  <xsl:template match="to">
    <xsl:text>TO: </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>
</xsl:text>
  </xsl:template>
  <xsl:template match="from">
    <xsl:text>FROM: </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>
</xsl:text>
  </xsl:template>
  <xsl:template match="message">
    <xsl:text>MESSAGE: </xsl:text>
    <xsl:apply-templates/>
    <xsl:text>
</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Notice how each template now outputs its label (e.g. `TO:`) followed by a single space, then finishes off with a line break. All the other whitespace in the style sheet is ignored, since it isn't mixed with text. This gives us the fine control over formatting that we need when outputting a plain text file.

Are we done yet? Not quite. We have to go back and add to our XML document a *style sheet declaration* that will point to our XSL file, just like we did for the CSS example. Simply open the XML document and insert the following line before the opening `<letter>` element:

Example 2.10. `letter-text.xml` (excerpt)

```
<?xml-stylesheet type="text/xsl" href="letter2text.xsl"
    version="1.0"?>
```

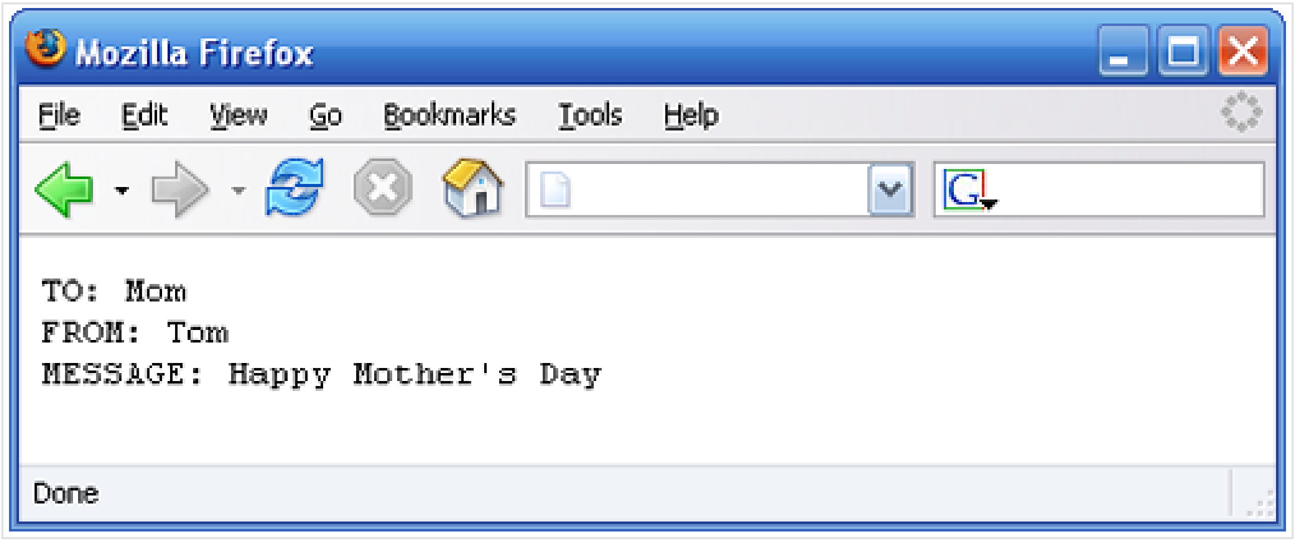
Now, our XML document looks like this:

Example 2.11. `letter-text.xml`

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="letter2text.xsl"
    version="1.0"?>
<letter>
  <to>Mom</to>
  <from>Tom</from>
  <message>Happy Mother's Day</message>
</letter>
```

When you view the XML document in Firefox, you should see something similar to the result pictured in Figure 2.2, “Viewing XSL results in Firefox.”. You can try viewing this in Internet Explorer as well, but you won’t see the careful text formatting we applied in our style sheet. Internet Explorer interprets the result as HTML code, even when the style sheet clearly specifies that it will output text. As a result, whitespace is collapsed and our whole document appears on one line.

Figure 2.2. Viewing XSL results in Firefox.



```
TO:  Mom
FROM:  Tom
MESSAGE: Happy Mother's Day
```

View larger image.

If you’re curious, go ahead and view the source of this document. You’ll notice that you won’t see the output of the transformation (technically referred to as the *result tree*), but you *can* see the XML document source.

What About my Favorite Browser?

If you don’t use Firefox on a regular basis, you might be a little miffed that I’ve started out with an example that works only in Mozilla-based browsers.

First of all, if you prefer Internet Explorer, the situation will improve with the next example, which conforms to Internet Explorer’s assumption that the result of a transformation must be HTML, not plain text as it was in this example.

As for the other browsers in popular use, including Safari and Opera, these do not yet support XSLT. For this reason, it is not yet practical to rely on browser support for XSLT in a real-world website. As we’ll learn, it is far more sensible to use XSLT on the server side, where it is safe from browser incompatibilities.

For now, however, the solid XSLT capabilities built into Firefox (and to a lesser degree, Internet Explorer) provide a convenient means to learn what XSLT is capable of.

Transforming XML into HTML

That wasn't so bad, was it? You successfully transformed a simple XML document into flat ASCII text, and even added a few extra tidbits to the output.

Now, it's time to make things a little more complex. Let's transform the XML document into HTML. Here's the great part – you won't have to touch the original XML document (aside from pointing it at a new style sheet, that is). All you'll need to do is create a new XSL file:

Example 2.12. `letter2html.xsl`

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/letter">
    <html>
      <head><title>Letter</title></head>
      <body><xsl:apply-templates/></body>
    </html>
  </xsl:template>
  <xsl:template match="to">
    <b>TO: </b><xsl:apply-templates/><br/>
  </xsl:template>
  <xsl:template match="from">
    <b>FROM: </b><xsl:apply-templates/><br/>
  </xsl:template>
  <xsl:template match="message">
    <b>MESSAGE: </b><xsl:apply-templates/><br/>
  </xsl:template>
</xsl:stylesheet>
```

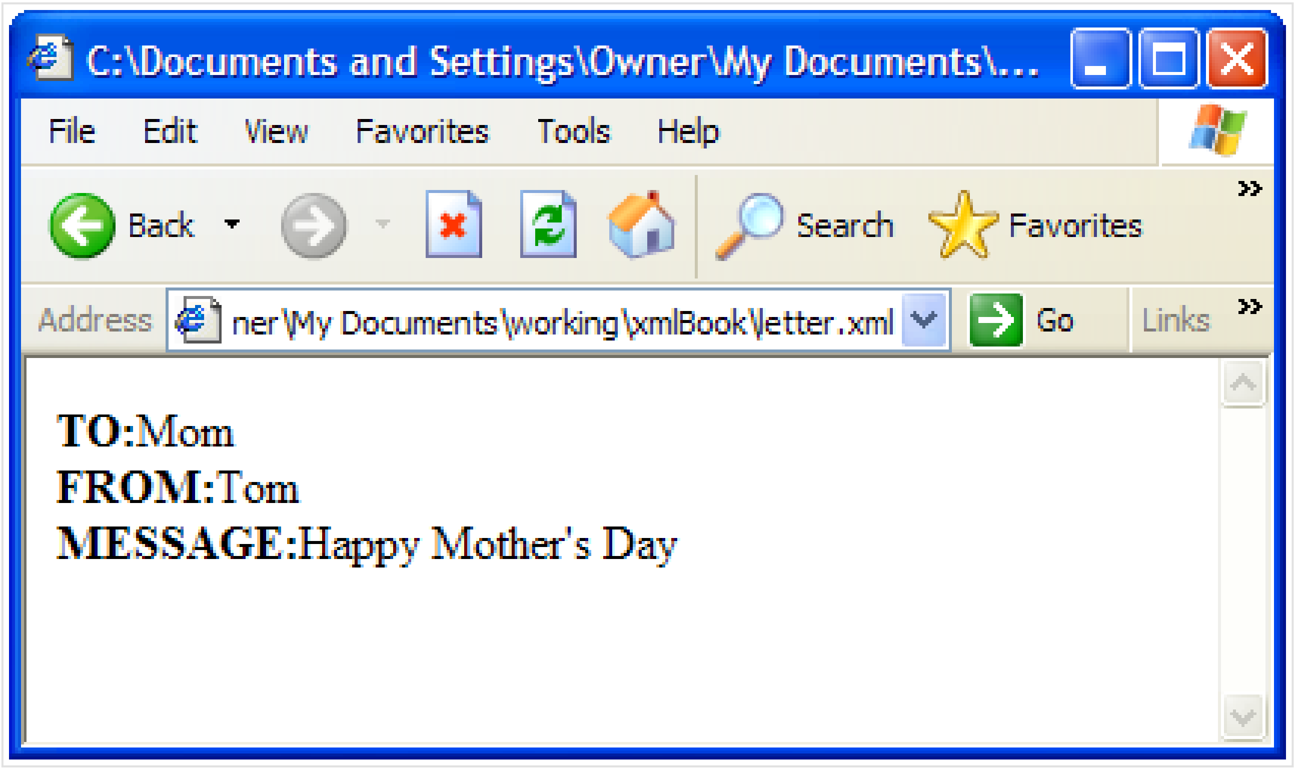
Right away, you'll notice that the style sheet's output element now specifies an output method of `html`. Additionally, our first template now outputs the basic tags to produce the framework of an HTML document, and doesn't bother suppressing the whitespace in the source document with a `select` attribute.

Other than that, these instructions don't differ much from our text-only style sheet. In fact, the only other changes we've made have been to tag the label for each line to be bold, and end each line with an HTML line break (`<br/>`). We no longer need the `<xsl:text>` tags, since our HTML `<b>` and `<br/>` tags perform the same function. Note the space following each label, which is inside the <b> tag so that it won't be ignored by the processor.

All we have to do now is edit our XML file to make sure that the `<?xml-stylesheet?>` instruction references our new style sheet (`letter-html.xml` in the code archive), and we're ready to display the results in a Web browser.

You should see something similar to Figure 2.3, "Viewing XSL Results in Internet Explorer.".

Figure 2.3. Viewing XSL Results in Internet Explorer.


C:\Documents and Settings\Owner\My Documents\...
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Address
ner\My Documents\working\xmlBook\letter.xml
Go
Links

TO: Mom
FROM: Tom
MESSAGE: Happy Mother's Day

View larger image.

Using XSLT to Transform XML into other XML

What happens if you need to transform your own XML document into an XML document that meets the needs of another organization or person? For instance, what if our letter document, which uses `<to>` , `<from>` , and `<message>` tags inside a `<letter>` tag, needed to have different names, say `<recipient>` , `<sender>` , and `<body>` ?

Not to worry – XSLT will save the day! And, as with the two previous examples, we don’t even need to worry about changing the source XML document. All we have to do is create a new XSL file, and we’re set.

As before, we’ll open with the standard `stylesheet` element, but, this time, we’ll choose `xml` as our output method. We’re also going to instruct XSLT to indent the resulting XML:

Example 2.13. `letter2xml.xsl` (excerpt)

```
 <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>
```

The `<template>` elements are structured as before, but this time they output the new XML elements:

Example 2.14. `letter2xml.xsl` (excerpt)

```
 <xsl:template match="/letter">

<letter><xsl:apply-templates/></letter>

</xsl:template>

<xsl:template match="to">

<recipient><xsl:apply-templates/></recipient>

</xsl:template>

<xsl:template match="from">

<sender><xsl:apply-templates/></sender>

</xsl:template>

<xsl:template match="message">
```

<**body**><xsl:apply-templates/></**body**>

</xsl:template>

</xsl:stylesheet>

Now, all you have to do is edit your XML document to point to the style sheet, and you'll be able to view your new XML in any Web browser, right? Wrong! You see, Web browsers only supply collapsible tree formatting for XML documents without style sheets. XML documents that result from a style sheet transformation are displayed without any styling at all, or at best are treated as HTML – not at all the desired result.

Where the browser *can* be useful for viewing XML output is when that XML is an XHTML document – which browsers obviously can display. There are several things that need to be added to your style sheet to signal to the browser that the document is more than a plain XML file, though. The first is the XHTML namespace:

Example 2.15. `letter2xhtml.xsl` **(excerpt)**

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
```

Here we have declared a default namespace for tags without prefixes in the style sheet. Thus tags like `<html>` and `<b>` will be correctly identified as XHTML tags.

Next up, we can flesh out the `output` element to more fully describe the output document type:

Example 2.16. `letter2xhtml.xsl` **(excerpt)**

```
<xsl:output method="xml" indent="yes" omit-xml-declaration="yes"
```

media-type="application/xhtml+xml" encoding="iso-8859-1"

doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"

doctype-system=

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

In addition to the `method` and `indent` attributes, we have specified a number of new attributes here:

`omit-xml-declaration`
This tells the processor not to add a `<?xml?>` declaration to the top of the output document. Internet Explorer for Windows displays XHTML documents in Quirks Mode when this declaration is present, so by omitting it we can ensure that this browser will display it in the more desirable Standards Compliance mode.

`media-type`
Though not required by current browsers, setting this attribute to `application/xhtml+xml` offers another way for the browser to identify the output as an XHTML document, rather than plain XML.

`encoding`
Sets the character encoding of the output document, controlling which characters are escaped as character references (`&xnn;`).

`doctype-public, doctype-system`
Together, these two attributes provide the values needed to generate the DOCTYPE declaration for the output document. In this example, we've specified values for an XHTML 1.0 Transitional document, but you could also specify an XHTML 1.0 Strict document if that's what you need:

```
 <xsl:output method="xml" indent="yes" omit-xml-declaration="yes"

media-type="application/xhtml+xml" encoding="iso-8859-1"

doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"

doctype-system=

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>
```

The rest of the style sheet is as it was for the HTML output example we saw above. Here's the complete style sheet so you don't have to go searching:

Example 2.17. `letter2xhtml.xsl`

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
```

```
  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes"
      media-type="application/xhtml+xml" encoding="iso-8859-1"
      doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
      doctype-system=
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

  <xsl:template match="/letter">
    <html>
      <head><title>Letter</title></head>
      <body><xsl:apply-templates/></body>
    </html>
  </xsl:template>

  <xsl:template match="to">
    <b>TO: </b><xsl:apply-templates/><br/>
  </xsl:template>

  <xsl:template match="from">
    <b>FROM: </b><xsl:apply-templates/><br/>
  </xsl:template>

  <xsl:template match="message">
    <b>MESSAGE: </b><xsl:apply-templates/><br/>
  </xsl:template>
</xsl:stylesheet>
```

Point the `<?xml-stylesheet?>` processing instruction in your XML document at this style sheet and then load it in Firefox or Internet Explorer. You should see the output displayed as an XHTML document.

So yes, if the XML you are generating happens to be XHTML, a browser can display it just fine. Otherwise, what we need to display XML output is some kind of standalone XSLT processor that we can run instead of a Web browser... but, guess what? We've run out of space to talk about XSLT in this chapter. We'll pick up this discussion in Chapter 4, *Displaying XML in a Browser*.

Our CMS Project

In Chapter 1, *Introduction to XML*, we did quite a bit of work to analyze the article content type. Now, we need to identify exactly what we need for our news items, binary files, and Web copy. We must also manage and track site administrators using XML. By the time we get to the end of this chapter, we'll be roughly twothirds the way through the requirements-gathering phase. Don't worry, though – time spent in this part of the process will pay off in a big way when we start development.

News

Compared to our article content type, news will be fairly straightforward. We will need to track these pieces of information:

- Unique identifier
- Headline
- Author
- Short description
- Publication date
- Status
- Keywords
- URL for more information

Everything else should look just like the article content type, except that we won't allow HTML tags inside our description. Here's what a typical news item would look like:

```
<news id="123">
  <headline>New XML application being built</headline>
  <author>Tom Myer</author>
  <description>A new XML application is now finally being released
    by ...</description>
  <pubdate>2004-01-20</pubdate>
  <status>live</status>
  <keywords>XML</keywords>
  <url>http://www.yahoo.com/</url>
</news>
```

From a programmatic standpoint, we will only display news pieces with a "live" status.

Web Copy

Many of our site's Web pages, including the homepage, will display copy of some form, be it the contact details for our company, or a description of the services we can provide. If we built a CMS that didn't allow us to manage this copy, we wouldn't have a proper CMS, would we?

The easiest way to keep track of copy is to treat each piece a little like an article. In fact, Web copy has many of the same characteristics as your standard articles, except that we generally don't need to track authors. An XML document that tracks a piece of Web copy will look like this:

```
<webcopy id="123">
  <navigationlabel>XML CMS</navigationlabel>
  <headline>XML-powered CMS Solutions</headline>
  <description>Learn about our XML-powered CMS products.
  </description>
  <pubdate>2004-01-20</pubdate>
  <status>live</status>
  <keywords>XML CMS</keywords>
  <body>[CDATA[
    <h1>Creating an XML-powered CMS</h1>
    <p>Are you tired of waiting around for your "IT Guy" or
      expensive designer to update your web site? Well, those
      days will be long forgotten if you buy our XML-powered CMS!
      With this revolutionary new tool, you can make quick and
      easy updates to your own web site! Forget all the hassles!
      It slices, it dices!</p>
  ]]</body>
</webcopy>
```

The `<keywords>` and `<status>` elements will work in much the same way as they do for articles and news pieces.

Administrators

Our final content type isn't really a content type – it's more of a supporting type. We will need to keep track of each administrator on the site, as these are the folks who can log in and make changes to advertisement copy, articles, news pieces, and binary files.

We will need to record each administrator's name, username, password (encrypted, of course), and email address. For the moment, we won't worry about exactly how the password is encrypted – we'll talk about that later.

Example 2.18. `admin.xml`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<admins>
  <admin id="1">
```

```
   <name>Joe</name>
   <username>joe</username>
   <password>$1$064.HQ..$x912OhlIlHFylTPJmJR/k/</password>
   <email>joe@myerman.com</email>
 </admin>
 <admin id="2">
   <name>Bill</name>
   <username>bill</username>
   <password>$1$Ep5.7h4.$R6iGqy.Wj2Dz8SAE9WG3l0</password>
   <email>bill@myerman.com</email>
 </admin>
 <admin id="3">
   <name>Tom</name>
   <username>tom</username>
   <password>$1$Cl/.j3..$QcjxGtxqYx0VNp3QanGnP0</password>
   <email>tom@myerman.com</email>
 </admin>
</admins>
```

As with each article/news item/binary file/advertisement copy item, each administrator will need a unique ID – otherwise, the system may not know who's trying to log in.

Summary

We covered a lot in this chapter – I'm glad you're still with me! In Chapter 3, *DTDs for Consistency*, we're going to dig around inside DTDs and XML Schemas. And, in the CMS section, we'll take a look at an alternative approach to handling status, keyword, and author listings – I think you'll really like the way we change things around. After that, you should have enough of a working knowledge of XML (and its wacky family) to really start development.

Chapter 3. DTDs for Consistency

So far, we've created some very simple XML documents and learned what they're made of. We've also walked through some very simple examples in which we've transformed XML into something else, be it text, HTML, or different XML. Now, it's time to learn how to make your XML documents consistent.

Consistency in XML

Ralph Waldo Emerson, the great American thinker and essayist, once said, "A foolish consistency is the hobgoblin of little minds." Well, foolish or not, in the world of XML, we like consistency. In fact, in many contexts, consistency can be a very beautiful thing.

Remember that XML allows you to create any kind of language you want. We've already seen some varying examples in this book: from a letter to mom, to articles and news stories. In many cases, as long as you follow the rules of well-formedness, just about anything goes in XML.

However, there will come a time when you need your XML document to follow some rules – to pass a validity test – and those times will require that your XML data be consistently formatted. For example, our CMS should not allow a piece of data that's supposed to be in the admin information file to show up in a content file. What we need is a way to enforce that kind of rule.

In XML, there are two ways to set up consistency rules: DTDs and XML Schema. A DTD (document type definition) is a tried and true (if not old-fashioned) way of achieving consistency. It has a peculiar, non-XML syntax that many XML newcomers find rather limiting, but which evokes a comfortable, hometown charm among the old-school XML programmers. XML Schema is newer, faster, better, and so on; it does a lot more, and is written like any other XML document, but many find it just as esoteric as DTDs.

Information on DTDs and XML Schema could fill thick volumes if we gave it a chance. Each of these technologies contains lots of hidden nooks and crannies crammed with rules, exceptions, notations, and side stories. But, remember why we're here: we must learn as much as we need to know, then apply that knowledge as we build an XML-powered Website.

Fun with Terminology

Speaking of side stories, did you know that DTD actually stands for two things? It stands not just for document type definition, but also document type declaration. The *declaration* consists of the lines of code that make up the *definition*. Since the distinction is a tenuous one, we'll just call them both "DTD" and move on!

This chapter will focus on DTDs, as you're still a beginner, and providing information on XML Schema would be overkill. However, I will take a few minutes to explain XML Schema at a high level, and provide some comparisons with DTDs.

Just a warning before we start this chapter: consistency in XML is probably the hardest aspect we've covered so far, because DTDs can be pretty esoteric things. However, I think you'll find it worth your while, since using a DTD will prevent many problems down the road.

What's the Big Deal About Consistency?

Okay, before we get started, let's ask a very obvious question: "Why, oh why, are we sitting here on a lovely Saturday afternoon talking about the importance of consistency in XML documents? Why aren't we out in the park with our loyal dog Rover, a picnic basket, and our wonderful significant other?"

Well, you've actually asked two questions there. I can't answer the second one, because I really don't want to get into your personal life right now. As for the first question, many possible answers spring to mind:

1. There will be a pop quiz later, so you'd better know your stuff.
2. Your boss told you to learn it.
3. You need to share your XML document with another company/department/organization, and they expect your information in a certain format.
4. Your application requires that the XML documents given to it pass certain tests.

Although answers 1 and 2 can loom large in one's life, answers 3 and 4 are more solid reasons to understand the importance of consistency in XML documents. Using a system to ensure consistency allows your XML documents to interact with all kinds of applications, contexts, and business systems – not just your own. In layman's terms, using a DTD with your XML documents makes them easier to share with the outside world.

DTDs

The way DTDs work is relatively simple. If you supply a DTD along with your XML file, then the XML parser will compare the content of the document with the rules that are set out in the DTD. If the document doesn't conform to the rules specified by the DTD, the parser raises an error and indicates where the processing failed.

DTDs are such strange creatures that the best way to describe them is to just jump right in and start writing them, so that's exactly what we're going to do. A DTD might look something like this:

```
<!DOCTYPE letter [
  ELEMENT letter (to,from,message)
  ELEMENT to (#PCDATA)
  ELEMENT from (#PCDATA)
  ELEMENT message (#PCDATA)
]>
```

Those of you who are paying attention should have noticed some remarkable similarities between this DTD and the Letter to Mother example that we worked on in Chapter 2, *XML in Practice*. In fact, if you look closely, each line of the DTD provides a clue as to how our letter should be structured.

The first line of the DTD, which begins with `<!DOCTYPE`, indicates that our document type is `letter`. Any document we create on the basis of this DTD must therefore have a `letter` as its root element, or the document won't be valid.

The rest of the DTD is devoted to explaining two things:

1. The proper order of elements in the XML document.
2. The proper content of elements in the XML document.

In the next few sections, I'll walk you through the most important parts of element declarations. Then, we'll work on attribute and entity declarations. Once we have all that under our belts, we'll get our hands dirty building some sample XML files with DTDs.

Element Declarations

Let's have a look at the next line of the DTD above: the one that comes after the DOCTYPE.

```
ELEMENT letter (to,from,message)
```

This is called an *element declaration*. You can declare elements in any order you want, but they must all be declared in the DTD. To keep things simple, though, and to mirror the order in which elements appear in the actual XML file, I'd suggest that you do what we've done here: declare your root element first.

A DTD element declaration consists of a tag name and a definition in parentheses. These parentheses can contain rules for any of:

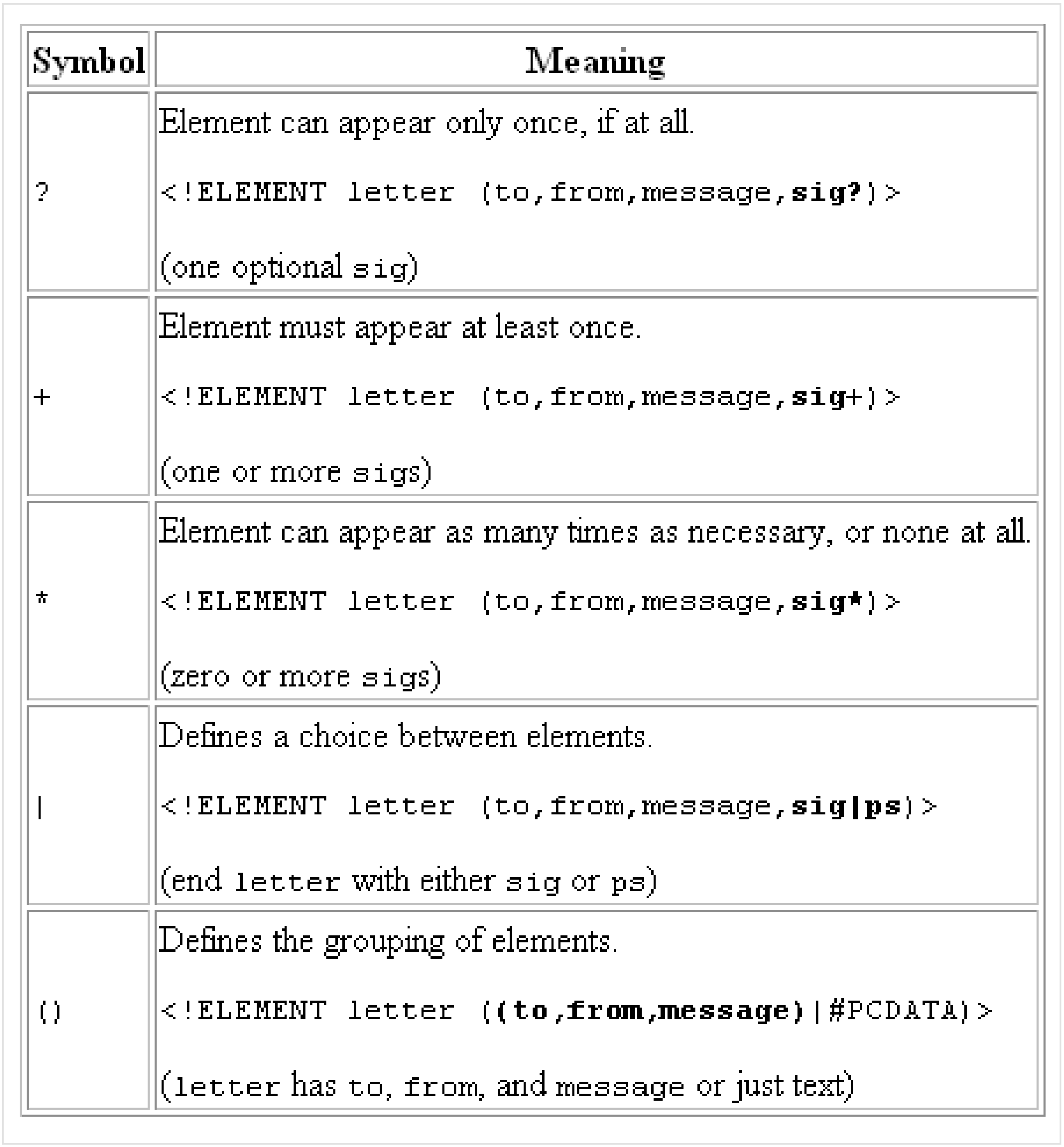
- Plain text
- A single child element
- A sequence of elements

In this case, we want the `letter` element to contain, in order, the elements `to`, `from`, and `message`. As you can see, the sequence of child elements is comma-delimited.

In fact, to be more precise, the sequence not only specifies the order in which the elements should appear, but also, how many of each element should appear. In this case, the element declaration specifies that one of each element must appear in the sequence. If our file contained two from elements, for example, it would be as invalid as if it listed the `message` element before `to`.

Naturally, there will come a time when you'll need to specify more than just one of each element. How will you do that? With a neat little system of notation, defined in Table 3.1, "XML Element Declaration Notation", which may remind you of UNIX regular expressions.

Table 3.1. XML Element Declaration Notation



Symbol	Meaning
?	Element can appear only once, if at all. `<!ELEMENT letter (to,from,message,`**`sig?`**`)>` (one optional `sig`)
+	Element must appear at least once. `<!ELEMENT letter (to,from,message,`**`sig+`**`)>` (one or more `sigs`)
*	Element can appear as many times as necessary, or none at all. `<!ELEMENT letter (to,from,message,`**`sig*`**`)>` (zero or more `sigs`)
|	Defines a choice between elements. `<!ELEMENT letter (to,from,message,`**`sig|ps`**`)>` (end `letter` with either `sig` or `ps`)
()	Defines the grouping of elements. `<!ELEMENT letter (`**`(to,from,message)`**`|#PCDATA)>` (`letter` has `to`, `from`, and `message` or just text)

With this notation as a backdrop, you can get pretty creative:

- Require at least two instances of an element.< code>ELEMENT chapter (title,para,para+)

  (at least two paras)
- Apply element count modifiers to element groups. `ELEMENT chapter ((title,para+)+)`

  (one or more `titles` , each followed by one or more `paras`)
- Allow an element to contain an element or plain text. `ELEMENT title (subtitle|#PCDATA)`

(`title` contains a `subtitle` or plain text)

- Require exactly three instances of an element. `ELEMENT instruction (step,step,step)`
  (exactly three `steps`)

Elements that Contain only Text

Let’s keep looking at our original DTD. After the `letter` declaration, we see these three declarations:

```
ELEMENT to (#PCDATA)
 ELEMENT from (#PCDATA)
 ELEMENT message (#PCDATA)
```

Here, we see `#PCDATA` used to define the contents of our elements. `#PCDATA` stands for parsed character data, and refers to anything other than XML elements. So whenever you see this notation in a DTD, you know that the element must contain only text.

Mixed Content

What if you want to have something like this in your XML document?

```
<paragraph>This is a paragraph in which items are <b>bolded</b>,
  <i>italicized</i>, and even <u>underlined</u>. Some items are
  even deemed <highpriority>high priority</highpriority>.
 </paragraph>
```

You’d probably think that you needed to declare the paragraph element as containing a sequence of `#PCDATA` and other elements, like this:

```
ELEMENT paragraph (#PCDATA,b,i,u,highpriority)  <!-- wrong! -->
```

You might think that, but you’d be wrong! The proper way to declare that an element can contain mixed content is to separate its elements using the `|` symbol and add a `*` at the end of the element declaration:

```
ELEMENT paragraph (#PCDATA|b|i|u|highpriority)* <!-- right! -->
```

This notation allows the `paragraph` element to contain any combination of plain text and `b`, `i`, `u`, and `highpriority` elements. Note that with mixed content like this, you have no control over the number or order of the elements that are used.

Empty Elements

What about elements such as the `hr` and `br`, which in HTML contain no content at all? These are called empty elements, and are declared in a DTD as follows:

```
ELEMENT hr EMPTY
 ELEMENT br EMPTY
```

So far, most of this makes good sense. Let's talk about attribute declarations next.

Attribute Declarations

Remember attributes? They're the extra bits of information that hang around inside the opening tags of XML elements. Fortunately, attributes can be controlled by DTDs, using what's called an attribute declaration.

An attribute declaration is structured differently than an element declaration. For one thing, we define it with `!ATTLIST` instead of `|!ELEMENT`. Also, we must include in the declaration the name of the element that contains the attribute(s), followed by a list of the attributes and their possible values.

For example, let's say we had an XML element that contained a number of attributes:

```
<actor actorid="HF1234" gender="male" type="superstar">
  Harrison Ford</actor>
```

The element and attribute declarations for that element might look like this:

```
ELEMENT actor (#PCDATA)
 ATTLIST actor
  actorid ID #REQUIRED
  gender (male|female) #REQUIRED
  type CDATA #IMPLIED
```

The easiest attribute to understand is `type` – it contains CDATA, or character data. Basically, this attribute can contain any string of characters or numbers. Acceptable values for this attribute might be "superstar", "leading man", or even "dinosaur." As developers, we can't exert much control over what is placed in an attribute of type `CDATA` .

Do you see `#IMPLIED` right after `CDATA` ? In DTD-speak, this means that the attribute is optional. Don't ask why they didn't use `#OPTIONAL` – this legacy has been passed down from the days of SGML, XML's more complex predecessor.

Let's take a look at the `gender` attribute's definition. This attribute is `#REQUIRED` , so a value for it has to be supplied with every `actor` element. Instead of allowing any arbitrary text, however, the DTD limits the values to either `male` or `female` .

If, in our document, an `actor` element fails to contain a `gender` attribute, or contains a `gender` attribute with values other than `male` or `female` , then our document would be deemed invalid.

Let's look at the most complex attribute value in our example, then we'll stop talking about attribute and element declarations. The `actorid` attribute has been designated an ID. In DTD-speak, an ID attribute must contain a unique value, which is handy for product codes, database keys, and other identifying factors.

In our example, we want the `actorid` attribute to uniquely identify each actor in the list. The `ID` type set for the `actorid` attribute ensures that our XML document is valid if and only if a unique `actorid` is assigned to each actor.

Some other rules that you need to follow for IDs include:

- ID values must start with a letter or underscore.
- There can only be one ID attribute assigned to an element.

Incidentally, if you want to declare an attribute that must contain a *reference* to a unique ID that is assigned to an element somewhere in the document, you can declare it with the `IDREF` attribute type. We won't have any use for this attribute type in this book, however.

Entity Declarations

Back in Chapter 1, *Introduction to XML*, we talked a little bit about entities. An entity is a piece of XML code that can be used (and reused) in a document with an *entity reference*. For example, the entity reference `&lt;` is used to represent the `<` character, an XML built-in entity.

XML supports a number of built-in entities (among them `&lt;` , `&gt;` , `&quote;` and `&amp;`) that don't ever need to be declared inside a DTD. With entity declarations, you can define your own entities – something that I think you'll find very useful in your XML career.

There are different types of entities, including general, parameter, and external. Let's go over each very quickly.

General entities are basically used as substitutes for commonly-used segments of XML code. For example, here is an entity declaration that holds the copyright information for a company:

```
ENTITY copyright "© 2004 by Triple Dog Dare Media"
```

Now that we've declared this entity, we could use it in our documents like so:

```
<footer>&copyright;</footer>
```

When the parser sees `&copyright;` , an entity reference, it looks for its entity declaration and substitutes the text we've declared as the entity.

There are a couple of restrictions on entity declarations:

- Circular references are not allowed. The following is a no-no:
  ```
   ENTITY entity1 "&entity2; is a real pain to deal with!"

  ENTITY entity2 "Or so &entity1; would like you to believe!"
  ```
- We can't reference a general entity anywhere but in the XML document proper. For entities that you can use in a DTD, you need parameter entities.

Parameter entities are both defined and referenced within DTDs. They're generally used to keep DTDs organized and to reduce the typing required to write them. Parameter entity names start with the `%` sign. Here's an example of a parameter entity, and its use in a DTD:

```
ENTITY % acceptable "(#PCDATA|b|i|u|citation|dialog)*"
```

```
ELEMENT paragraph %acceptable;
ELEMENT intro %acceptable;
ELEMENT sidebar %acceptable;
ELEMENT note %acceptable;
```

What this says is that each of the elements `paragraph` , `intro` , `sidebar` , and `note` can contain regular text as well as `b` , `i` , `u` , `citation` , and `dialog` elements. Not only does the use of a parameter entity reduce typing, it also simplifies maintenance of the DTD. If, in the future, you wanted to add another element (`sidebar`) as an acceptable child of those elements, you'd only have to update the `%acceptable;` entity:

```
ENTITY % acceptable "(#PCDATA|b|i|u|citation|dialog|sidebar)"
```

External entities point to external information that can be copied into your XML document at runtime. For example, you could include a stock ticker, inventory list, or other file, using an external entity.

```
ENTITY favquotes SYSTEM "http://www.example.com/favstocks.xml"
```

In this case, we're using the `SYSTEM` keyword to indicate that the entity is really a file that resides on a server. You'd use the entity in your XML documents as follows:

```
<section>
  <heading>Current Favorite Stock Picks</heading>
  &favquotes;
</section>
```

External DTDs

The DTD example we saw at the start of this chapter appeared within the DOCTYPE declaration at the top of the XML document. This is okay for experimentation purposes, but with many projects, you'll likely have dozens – or even hundreds – of files that must conform to the same DTD. In these cases, it's much smarter to put the DTD in a separate file, then reference it from your XML documents.

An external DTD is usually a file with a file extension of `.dtd` – for example, `letter.dtd` . This external DTD contains the same notational rules set forth for an internal DTD.

To reference this external DTD, you need to add two things to your XML document. First, you must edit the XML declaration to include the attribute

`standalone="no"` :

```
<?xml version="1.0" standalone="no"?>
```

This tells a validating parser to validate the XML document against a separate DTD file. You must then add a `DOCTYPE` declaration that points to the external DTD, like this:

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

This will search for the `letter.dtd` file in the same directory as the XML file. If the DTD lives on a Web server, you might point to that instead:

```
<!DOCTYPE letter SYSTEM
    "http://www.example.com/xml/dtd/letter.dtd">
```

A 10,000-Foot View of XML Schema

The XML Schema standard fulfills the same requirements as DTDs: it allows you to control the structure and content of an XML document. But, if it serves the same purpose as DTDs, why would we use XML Schema?

Well, DTDs have a few disadvantages:

1. DTD notation has little to do with XML syntax, and therefore cannot be parsed or validated the way an XML document can.
2. All DTD declarations are global, so you can't define two different elements with the same name, even if they appear in different contexts.
3. DTDs cannot strictly control the type of information a given element or attribute can contain.

XML Schema is written in XML, so it can be parsed by an XML parser. XML Schema allows you, through the use of XML namespaces, to define different elements with the same name. Finally, XML Schema provides very fine control over the kinds of data contained in an element or attribute.

Now, for some major drawbacks: if you thought that DTDs were esoteric, then you won't be pleased by the complexity introduced by XML Schema. Most of the criticism aimed at XML Schema is focused on its complexity and length. In fact, at first glance, a schema's verbosity will remind you of your motor-mouth friend who hogs the airspace at any gathering.

We won't get much of a chance to work with XML Schema in this book, but there are many fine books available on the subject.

Getting Our Hands Dirty

Okay, now you know a lot more about DTDs than you did before. If you're thinking that all this talk of consistency in XML seems fairly esoteric, you're not alone. But stick with me – we're about to embark on the practical examples that will illustrate exactly how these concepts fit into the overall XML picture.

Let's start out by creating a sample document and using a DTD to validate it. For this exercise, we'll be working with Macromedia Dreamweaver MX, as it includes a built-in XML validator.

Our First Case: A Corporate Memo

You work for Amalgamated International, LLC. The big boss comes into your office because he heard a rumor that you're an XML wizard. This is really great news, because he's just come back from a conference where he learned that XML is a terrific way to get your internal corporate memos under control.

He instructs you to figure out how to get all the corporate memos into XML, and yes, they do need to be validated, because they will be used later by an application that's capable of searching through the memos.

The first thing you do is you take a look at the dozens of corporate memos you and your colleagues have received in the past few months. After a day or two of close examination, a pattern emerges.

Just by looking at them, you can see that all memos have the following elements:

- Date
- Sender
- Recipient list
- Priority
- Subject line
- One or more paragraphs
- Signature block

- Preparer's initials

You're sure that there's more to it than that, so you decide to gather more information. When you talk to your department's administrative assistant, he fills in the rest of the picture:

- There is almost always some kind of departmental code assigned to the file. This code is not always printed on the physical memos, but is always used as part of the filename. These codes help designate the memo's department of origin (accounting, finance, marketing, etc.).
- There is almost always a blind copy list on each memo – in other words, a list of recipients who, though they received it, are not listed anywhere on the memo as having received it.
- Many memos also have an expiration date. At Amalgamated, if a given memo has no expiration date, the information on the memo is deemed good for 180 days. Most memos contain information with lifetimes of less then six months, so most employees never see this kind of information. Other memos – those concerning HR policies, for instance – may have expiration dates that are years away.

With this information in hand, you begin to create a DTD for XML-based memos.

Although your first impulse might be to run out and create a sample XML memo document, please resist that urge for now. There's nothing wrong with this approach – indeed, it does provide useful modeling techniques. However, right now, we want to work with DTDs, then apply what we know to the building of the XML document.

So, the first thing you need to do is declare a DOCTYPE. Because these memos are internal to the company, and there may be a need for a separate external memo DOCTYPE, you decide to use `internalmemo` as your root element name:

Example 3.1. `internalmemo-standalone.xml` **(excerpt)**

```
<?xml version="1.0"?>
<!DOCTYPE internalmemo [
```

Now, it's time to define your elements. The first element – the root element – is `internalmemo`. This element will contain all the other elements, which hold date, sender, recipient, subject line, and all other information. Because these represent a lot of elements, it would be useful to split your document into two logical partitions: `header` and `body`. The `header` will contain recipient, subject line, date, and other information. The `body` will contain the actual text of the memo.

Here is the element declaration for our root element:

Example 3.2. `internalmemo-standalone.xml` **(excerpt)**

```
ELEMENT internalmemo (header,body)
```

In DTD syntax, the above declaration states that our `internalmemo` element must contain one `header` element and one `body` element. Next, we will indicate which elements these will contain.

Here's what the `header` will contain:

Example 3.3. `internalmemo-standalone.xml` **(excerpt)**

```
ELEMENT header (date,sender,recipients,blind-recipients?,
    subject)
```

In DTD syntax, the above declaration states that the `header` element must contain single `date`, `sender`, and `recipients` elements, an optional `blind-recipients` element, and then a `subject` element.

Here is the `body`:

Example 3.4. `internalmemo-standalone.xml` **(excerpt)**

```
ELEMENT body (para+,sig)
```

In DTD syntax, the above declaration states that the `body` element must contain one or more `para` elements, followed by a single `sig` element.

Most of the other elements will contain plain text, except the `para` elements, in which we will allow bold and italic text formatting.

Example 3.5. `internalmemo-standalone.xml` **(excerpt)**

```
ELEMENT date (#PCDATA)
 ELEMENT sender (#PCDATA)
 ELEMENT recipients (#PCDATA)
 ELEMENT blind-recipients (#PCDATA)
 ELEMENT subject (#PCDATA)
 ELEMENT sig (#PCDATA)
 ELEMENT para (#PCDATA|b|i)*
 ELEMENT b (#PCDATA)
 ELEMENT i (#PCDATA)
```

That was simple enough. However, when we glance at the requirements, we can see that we haven't even begun to handle priority levels, preparer's initials, expiration dates, and department of origin.

What's the best way to handle these pieces of information? We could certainly add them as elements in the `head` section of our memos, but that wouldn't make much sense. Those pieces of information are hardly ever displayed on a document – they are used only for administrative purposes.

In any case, we want to be able to control the data that document creators put in for values such as priority. It wouldn't make much sense for them to enter "alligator" or "Disney World" when our application is going to be looking for "low", "medium" and "high."

The best way to store these pieces of information is to add them as attributes to the root element. To do that, we need to add an attribute declaration to our DTD:

Example 3.6. `internalmemo-standalone.xml` **(excerpt)**

```
ATTLIST internalmemo
  priority (low|medium|high) #REQUIRED
  initials CDATA #REQUIRED
  expiredate CDATA #REQUIRED
  origin (marketing|accounting|finance|hq|sales|ops) #REQUIRED
 ]>
```

So, what does a valid internal memo document look like? I'm glad you asked:

Example 3.7. `internalmemo-standalone.xml`

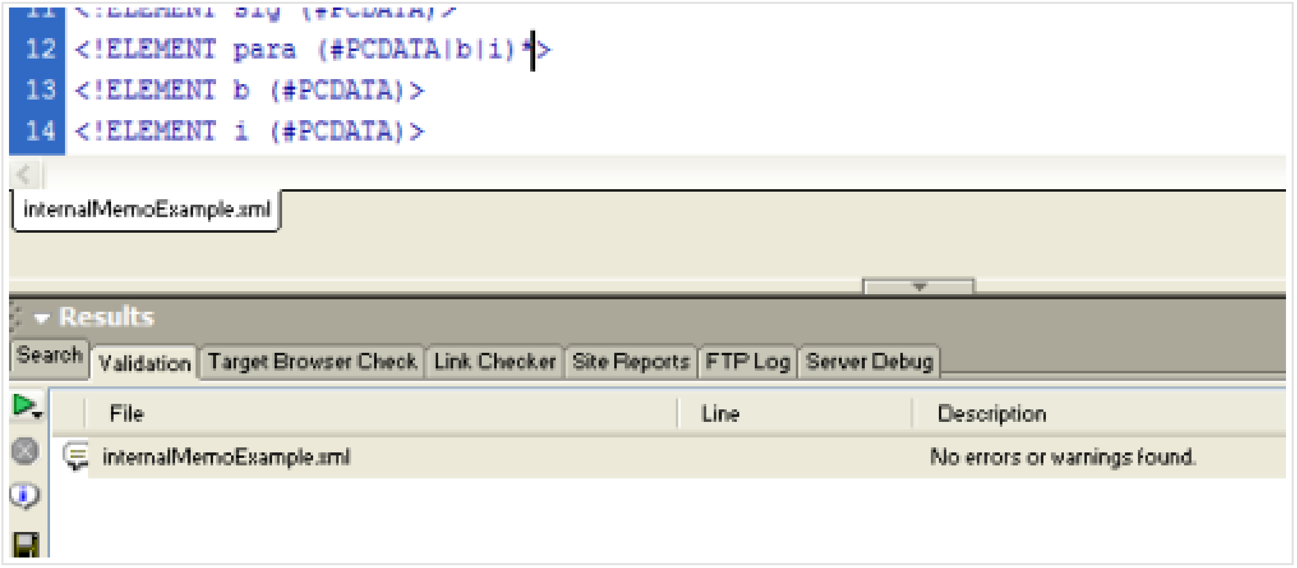
```
<?xml version="1.0"?>
 <!DOCTYPE internalmemo [
 ELEMENT internalmemo (header,body)
 ELEMENT header (date,sender,recipients,blind-recipients?,
    subject)
 ELEMENT body (para+,sig)
 ELEMENT date (#PCDATA)
 ELEMENT sender (#PCDATA)
 ELEMENT recipients (#PCDATA)
 ELEMENT blind-recipients (#PCDATA)
 ELEMENT subject (#PCDATA)
 ELEMENT sig (#PCDATA)
 ELEMENT para (#PCDATA|b|i)*
 ELEMENT b (#PCDATA)
 ELEMENT i (#PCDATA)
 ATTLIST internalmemo
  priority (low|medium|high) #REQUIRED
  initials CDATA #REQUIRED
  expiredate CDATA #REQUIRED
  origin (marketing|accounting|finance|hq|sales|ops) #REQUIRED
 ]>
 <internalmemo priority="high" initials="hjd"
    expiredate="01/01/2008" origin="marketing">
  <header>
    <date>01/05/2004</date>
    <sender>Thomas Myer</sender>
    <recipients>Marketing Department</recipients>
    <subject>Sell more stuff</subject>
  </header>
  <body>
    <para>This is a <i>simple</i> memo from the marketing
 department: sell <b>more</b> stuff!</para>
    <sig>Thomas Myer</sig>
  </body>
 </internalmemo>
```

Validating Our First Case

Now that we have a DTD and XML document, it's time to validate. Fortunately, Macromedia Dreamweaver MX has a built-in validation tool that we can use during development (in "real life" we would use a built-in validator that's part of our application). If you don't already own Dreamweaver, you can get a trial copy.

All we have to do is open our XML document (which contains a DTD) in Dreamweaver, then choose File > Check Page > Validate as XML. The result should look a lot like Figure 3.1, "Validating our first case with Dreamweaver MX.".

Figure 3.1. Validating our first case with Dreamweaver MX.



```
<!ELEMENT sig (#PCDATA)>
<!ELEMENT para (#PCDATA|b|i)*>
<!ELEMENT b (#PCDATA)>
<!ELEMENT i (#PCDATA)>
```

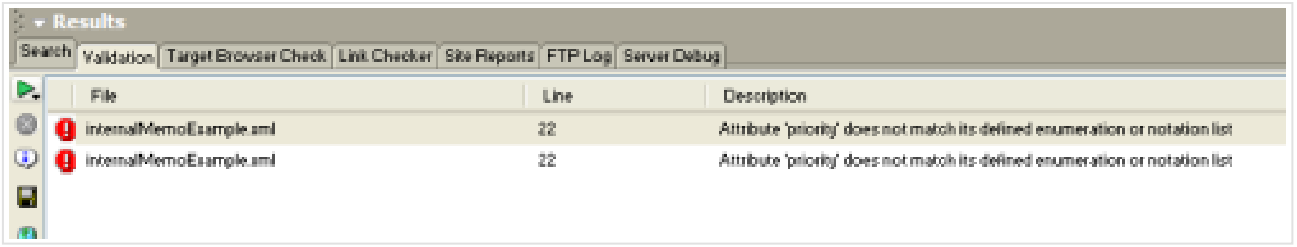
File	Line	Description
internalMemoExample.xml		No errors or warnings found.

View larger image.

Do you see how, under Results, it reads No errors or warnings found.? That's what you want to see. In Dreamweaver MX 2004, the results list for a valid document is simply empty, and the status bar beneath the list reads Complete.

What happens if some things are out of place? For instance, what if, as a priority, you wrote "Extremely Urgent"? What would happen then? In that case, you'd see an error message like the one in Figure 3.2, "Error resulting from a bad attribute value." below.

Figure 3.2. Error resulting from a bad attribute value.



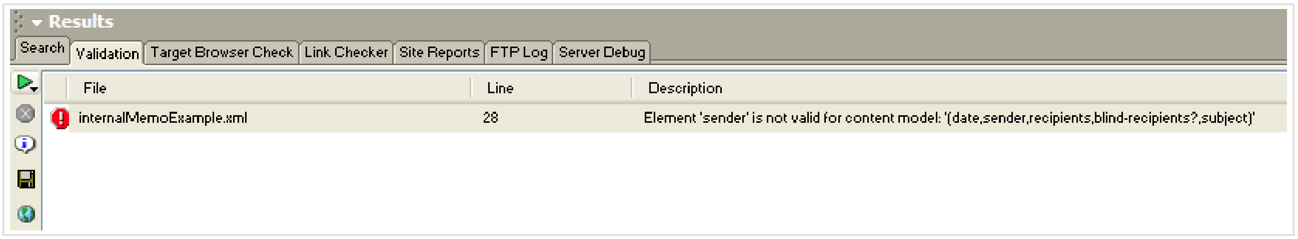
File	Line	Description
internalMemoExample.xml	22	Attribute 'priority' does not match its defined enumeration or notation list
internalMemoExample.xml	22	Attribute 'priority' does not match its defined enumeration or notation list

View larger image.

Notice that Dreamweaver MX tells you where the problem lies (with a specific line number) and provides a description of the problem. In this case, the validator is saying that the value of the priority attribute in your XML document doesn't match any of the possibilities defined in the DTD.

What if you decided to put the `<sender>` tag before the `<date>` tag? The validator catches that too, as you can see in Figure 3.3, “Error resulting from a misplaced element.”.

Figure 3.3. Error resulting from a misplaced element.



File	Line	Description
internalMemoExample.xml	28	Element 'sender' is not valid for content model: '(date,sender,recipients,blind-recipients?,subject)'

View larger image.

Again, the validator gives you a line number and a description that can lead you to resolve the problem. All you need to do is put the `sender` element back in the prescribed order, and the document will validate once more.

Second Case: Using an External DTD for Memos

Our first case was simple enough – an internal memo DTD and XML file. In that case, we embedded the DTD right into the file. This is a practical thing to do when you're only dealing with a small number of files for each DTD, but in Amalgamated's case, they'll be dealing with tens (if not hundreds) of thousands of memos.

There's no way that you want to have to maintain all those copies of the DTD separately. Instead, you want to have a single DTD that is included in all of your XML files. What you do is copy your DTD code out of your XML document and save it in a separate file called `internalmemo.dtd`. Don't copy the DOCTYPE line, or the last line that closes off the brackets!

When you're finished, your DTD file should look like this:

Example 3.8. `internalmemo.dtd`

```
LEMENT internalmemo (header,body)
 ELEMENT header (date,sender,recipients,blind-recipients?,
    subject)
 ELEMENT body (para+,sig)
 ELEMENT date (#PCDATA)
 ELEMENT sender (#PCDATA)
 ELEMENT recipients (#PCDATA)
 ELEMENT blind-recipients (#PCDATA)
 ELEMENT subject (#PCDATA)
 ELEMENT sig (#PCDATA)
 ELEMENT para (#PCDATA|b|i)*
 ELEMENT b (#PCDATA)
 ELEMENT i (#PCDATA)
 ATTLIST internalmemo
  priority (low|medium|high) #REQUIRED
  initials CDATA #REQUIRED
  expiredate CDATA #REQUIRED
  origin (marketing|accounting|finance|hq|sales|ops) #REQUIRED
```

Next, place a link to that external DTD in your XML document, like this:

Example 3.9. `internalmemo.xml` **(excerpt)**

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

You also need to change your XML document declaration (the first line of our XML document) to look like this:

Example 3.10. `internalmemo.xml` **(excerpt)**

```
<?xml version="1.0" standalone="no"?>
```

If you've done everything right, your file should validate when you use Dreamweaver's built-in validator. You now have a reusable DTD that you can apply to other internal memos.

Our CMS Project

In Chapter 2, *XML in Practice*, we added a few more content types to our CMS project. We now understand articles, news stories, binary files, and Web copy, and are well on our way to completing the requirements-gathering phase of the project – we can start coding soon!

However, and this is a big "however," we've also run into something of a problem. If you recall, we are tracking author, status, keyword, and other vital information in separate files. That is, each individual article, news story, binary file, and Web copy file keeps track of its own keywords, status, author, and dates.

For most of this information, which will rarely be used except in connection with the particular document, this isn't a problem, but author information is something of a special case. If we wanted to display all documents for a certain author, we would have to dig through all of our files to find all the matches. This isn't a big deal when our site is small, but the task grows more unmanageable with each passing day.

Never fear – I have a proposal that will solve this problem. In fact, the rest of this chapter will be devoted to tackling this issue. With any luck, it will also give you some insights into the ways in which you can analyze requirements and come up with more architecturally sound XML designs.

Reworking the Way we Track Author Information

Let's take a quick look at our article. I've reprinted what we came up with at the end of Chapter 1, *Introduction to XML* below for easy reference:

```
<article id="123">
  <author>Tom Myer</author>
  <headline>Creating an XML-powered CMS</headline>
  <description>This article will show you how to create an
    XML-powered content management system</description>
  <pubdate>2004-01-20</pubdate>
  <status>live</status>
  <keywords>XML CMS</keywords>
  <body>[CDATA[
    <h1>Creating an XML-powered CMS</h1>
    <p>In this article...</p>
  ]]</body>
</article>
```

So far, it's been very convenient to track our author information using the `author` element. However, doing it this way presents two problems, one of which we've already mentioned: eventually, we will have hundreds of articles on the site, and it would put a lot of strain on our application to dig through each one in order to display a list of articles by author.

The other problem is a little less obvious. What happens if, in one article, my name is listed as "Tom Myer," and in another, it's "Thomas Myer"? Or if, in one article, someone misspells my name as "Tom Meyer" (this happens a lot). To our application, these three names are different, and articles will thus be listed under three different authors.

To solve this problem, we should create a separate author listing (`authors.xml`), then use an `authorid` to reference that information in our articles. Once we have this figured out, we can get rid of the `author` element in all the other content types, and replace them with an `authorid` elements.

Handling our authors this way also allows us to track other information about authors, such as their email addresses, their bylines (in case they want to publish under pseudonyms), and other such information.

Here's a sample of what that code would look like:

Example 3.11. `authors.xml`

```
<authors>
  <author id="1">
    <name>Thomas Myer</name>
    <byline>myerman</byline>
    <email>tom@tripledogdaremedia.com</email>
  </author>
 </authors>
```

Instead of a separate `author` element, we would add an `authorid` element to our articles, like this:

```
<article id="123">
  <authorid>1</authorid>
  ...
```

Now we've solved the problem of *redundancy* – in other words, we've centralized our author information instead of having it spread across many different files. All we need to do is use this author ID in our articles, news stories, and all other content we add to our CMS; this ID is used to look up the author and retrieve the information we need.

Assign DTDs to our Project Documents?

The big question remains: do we take the time and effort to create DTDs or schemas for each of our content types? The answer is, as with most things technical, "it depends."

To be completely honest, most articles, news stories, and such will be submitted to the site through our administrative tool. This tool will have the necessary forms that will restrict data entry to certain fields. In other words, our administrative tool will do most of the work of validating our content. You could, therefore, suggest that a DTD would be completely superfluous, and you'd be right.

However, I think it would be good practice to develop a DTD for our article content type – after all, this is one of the most important document types we have in our system, and it has to be done right.

Here's a first shot at our article DTD:

```
 ELEMENT article (authorid,headline,description,pubdate,status,
    keywords,body)
 ATTLIST article
  id CDATA #REQUIRED
 ELEMENT authorid (#PCDATA)
 ELEMENT headline (#PCDATA)
 ELEMENT description (#PCDATA)
 ELEMENT pubdate (#PCDATA)
 ELEMENT status (#PCDATA)
 ELEMENT keywords (#PCDATA)
 ELEMENT body (#PCDATA)
```

Although we have declared our `body` element to contain character data, our article bodies will indeed be formatted using HTML tags. Because this HTML content will be wrapped in a CDATA block, those tags will be ignored by any XML processor reading an article file. We can use a CDATA block to hold any kind of text, as the XML parser will ignore any XML syntax that might appear in it. We therefore don't need to worry about the intricacies of HTML markup in this DTD.

If you asked ten XML folks whether they agreed with this approach, you'd get ten different opinions and alternative approaches. For now, we've created something that will work – and work quickly.

If you'd like more practice with DTDs, you can go back to Chapter 2, *XML in Practice* and look at the XML formats we created for our other content types, like Web copy and news items. Try writing DTDs for these as well. If you ever need to check the documents stored in your CMS for validity, you can use these DTDs to do it.

Summary

Wow! In three chapters we've covered basic XML, some XSLT and CSS, and, now, the basics of DTDs. Plus, we've nailed down most of the requirements for our CMS project. I think we're in pretty good shape to start looking more deeply at the rest of our project. Along the way, we'll pick up a few more XSLT and XML tricks.

Chapter 4. Displaying XML in a Browser

In Chapter 2, *XML in Practice*, we went over some basic XSLT and CSS using a very simple XML document. In this chapter, we're going to revisit some of those concepts with a more complex document. Once we've taken care of that, we'll return to our CMS project and start building the display pages for our site.

A Word on XPath

We've already been exposed to XSLT to a small degree. We used it to transform an XML letter to mother into something that could be displayed in a browser window. In this chapter, we're going to use a much more complex document as our starting point, and we'll learn how to use XPath.

Understanding XPath is the key to making effective use of XSLT. XPath is used in a variety of applications and technologies, however, XSLT is where its power and versatility really shine.

For all intents and purposes, XPath is a query language. It allows us to declaratively specify a "path" to an element or group of elements in an XML document. It uses a simple notation that is very similar to directory paths (hence the name XPath). You've already seen XPath in action within XSLT through some of the earlier examples.

When we put together a template, we normally use XPath to establish a match. For example, we can always handle the root of an XML document like this:

```
<xsl:template match="/">
```

With XPath, you can select all elements that have a particular tag name. For example, this template will match all the `<title>` tags in the document:

```
<xsl:template match="title">
```

Or, you could match certain elements depending on their location within an XML file. To match `<title>` tags that have a `<memo>` tag as their parent, you would use this expression:

```
<xsl:template match="memo/title">
```

As you can see, the basic XPath syntax looks a lot like a file path on your computer. That's because XML documents and your computer's file system are both hierarchical in nature. But you can go a step further and set conditions on which elements are matched within your specified path. These conditions are called *predicates*, and appear within square brackets following the element name you wish to set conditions for.

This example contains a predicate to make sure that it matches only `<title>` tags whose `priority` attribute is set to `hot` :

```
<xsl:template match="title[@priority='hot']">
```

The `@` symbol identifies `priority` in this example as an attribute name, not a tag name.

XPath also has a number of useful functions built in. For example, if you need to grab the first or last element of a series, you can use XPath to do so. This template will match the first `<para>` tag within each `<memo>` tag:

```
<xsl:template match="memo/para[first()]">
```

This template will match the first `<para>` tag within the last `<memo>` tag:

```
<xsl:template match="memo[last()]/para[first()]">
```

Although most practical applications are relatively simple, XPath can get quite twisty when it needs to be. The XPath Recommendation is quite a useful reference to these areas of complexity.

I've been giving you examples within an XSLT context, but XPath is used in a lot of different places, including PHP 5's new SimpleXML API. We'll get into SimpleXML a little later.

Practical XSLT Application

Instead of using a simple letter to mother, let's use something a bit more complex: a book chapter. Book chapters provide an excellent opportunity to understand the arbitrary complexity of most XML documents.

If you were to look at a typical book chapter (like this one), you'd probably only think of it as a flow of information. From the perspective of an XML document designer, however, a book chapter can be intimidatingly complex. Chapters can have titles and sections, and those sections can have titles. There are paragraphs throughout – some belong to the chapter (for example, introductory paragraphs), but others belong to sections. Sections can contain subsections. Paragraphs can contain text in italics, bold text, and other inline markup. In fact, one could even have different *types* of paragraphs, like notes, warnings, and tips. We mustn't forget that chapters can also hold non-textual content, in the form of images, graphs, and other visual materials. There are lots of possibilities for displaying these kinds of information.

Here's what a very short chapter might look like:

Example 4.1. `chapter.xml`

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="chapter2html.xsl"?>
<chapter id="example">
  <title>XML Example</title>
  <para type="intro">This is an introductory paragraph. It doesn't
    belong to any of the sections.</para>
  <section>
    <title>Main Section</title>
    <para type="intro">This is the <b>first</b> paragraph of the
      first section.</para>
    <para>Second paragraph.</para>
    <para type="note">This is a note!</para>
    <para type="warning">Don't even think about turning the page
      yet!</para>
    <section>
      <title>Subsection</title>
      <para type="intro">Looks like we started another section
        here!</para>
    </section>
  </section>
  <section>
    <title>Another Section</title>
    <para type="intro">And the chapter continues...</para>
  </section>
</chapter>
```

This sample file could go on and on, but I think you get the idea. Now it's time to try to parse this document and make sense of it. We'll perform some simple tasks first, then extend our knowledge as we go.

A First Attempt at Formatting

Now, let's create the corresponding XSL file, `chapter2html.xsl`. This file will contain all the instructions we will use to transform the XML elements in the chapter file we have just seen into XHTML. As we saw in Chapter 2, *XML in Practice*, an XSL file that generates XHTML should begin as follows:

Example 4.2. `chapter2xhtml.xsl` **(excerpt)**

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">

  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes"
    media-type="application/xhtml+xml" encoding="iso-8859-1"
    doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
    doctype-system=
```

Now, let's start matching elements. The first thing we want to do is to match the root of our document. We can use this template to output the basic tags required to produce an XHTML document:

Example 4.3. `chapter2xhtml.xsl` **(excerpt)**

```
<xsl:template match="/">
```

<html>

<head>

<title>A Book Chapter</title>

<meta http-equiv="content-type"

content="application/xhtml+xml; charset=iso-8859-1"/>

</head>

```
<body>

<xsl:apply-templates/>

</body>

</html>

</xsl:template>
```

Remember that, in XPath notation, `/` by itself stands for the root of your document, so we can rest assured that this template will only match once for each document that this style sheet transforms.

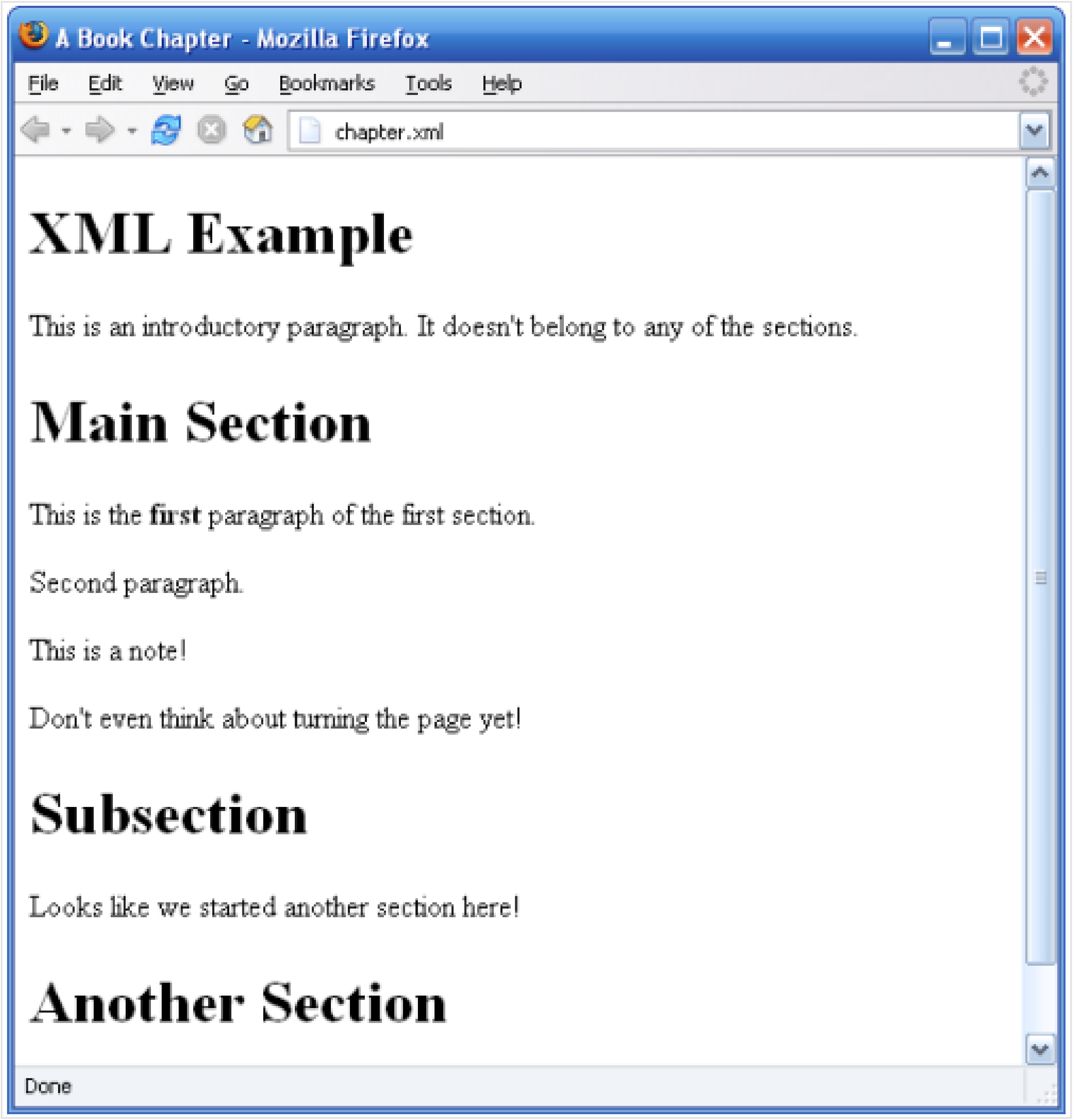
The `apply-templates` element then goes looking for other elements to match, so let's write some templates for those that it is likely to find. At this stage there's nothing we really want to output for the `chapter` element that we haven't already written out for the document root above, so we'll let the XSLT processor handle that with its default behavior for now. Let's instead concentrate on the elements inside the chapter:

Example 4.4. `chapter2html.xsl` **(excerpt)**

```
 <xsl:template match="title">

<h1><xsl:apply-templates/></h1>

</xsl:template>

<xsl:template match="para">

<p><xsl:apply-templates/></p>

</xsl:template>

<xsl:template match="b">

<b><xsl:apply-templates/></b>

</xsl:template>
```

Nothing could be simpler, right? We've matched all of our elements and for each we have output HTML tags as needed. Viewed in a browser, our output will look something like that shown in Figure 4.1, "Viewing the chapter example in Firefox."

Figure 4.1. Viewing the chapter example in Firefox.


A Book Chapter - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
chapter.xml
Done

XML Example

This is an introductory paragraph. It doesn't belong to any of the sections.

Main Section

This is the first paragraph of the first section.

Second paragraph.

This is a note!

Don't even think about turning the page yet!

Subsection

Looks like we started another section here!

Another Section

View larger image.

Looks pretty good, doesn't it? But, isn't there something missing? Of course there is. In our XSLT file, we are treating all `para` and `title` elements the same, regardless of where they appear in the XML document. That ain't right!

Using XPath to Discern Element Context

The `title` element near the top of the document is the chapter title, and should be handled differently from the `title` elements in the different nested sections. Likewise, `para` elements that denote warnings or introductions should be handled differently from other paragraphs.

Let's handle the `title` elements first. Chapter titles should be formatted with `<h1>` tags. Other `title` elements, which serve as nested section titles, should use incrementally smaller headings (`<h2>` , `<h3>` , and so on) in accordance with their level of nesting.

To distinguish between these different `title` types, you can use XPath notation. To pick out `title` elements that are children of the `chapter` tag, we can use the XPath expression `chapter/title` . To pick out `title` elements in top-level sections, we can use `chapter/section/title` , and so forth.

So here's an effective set of templates to handle the titles in our document:

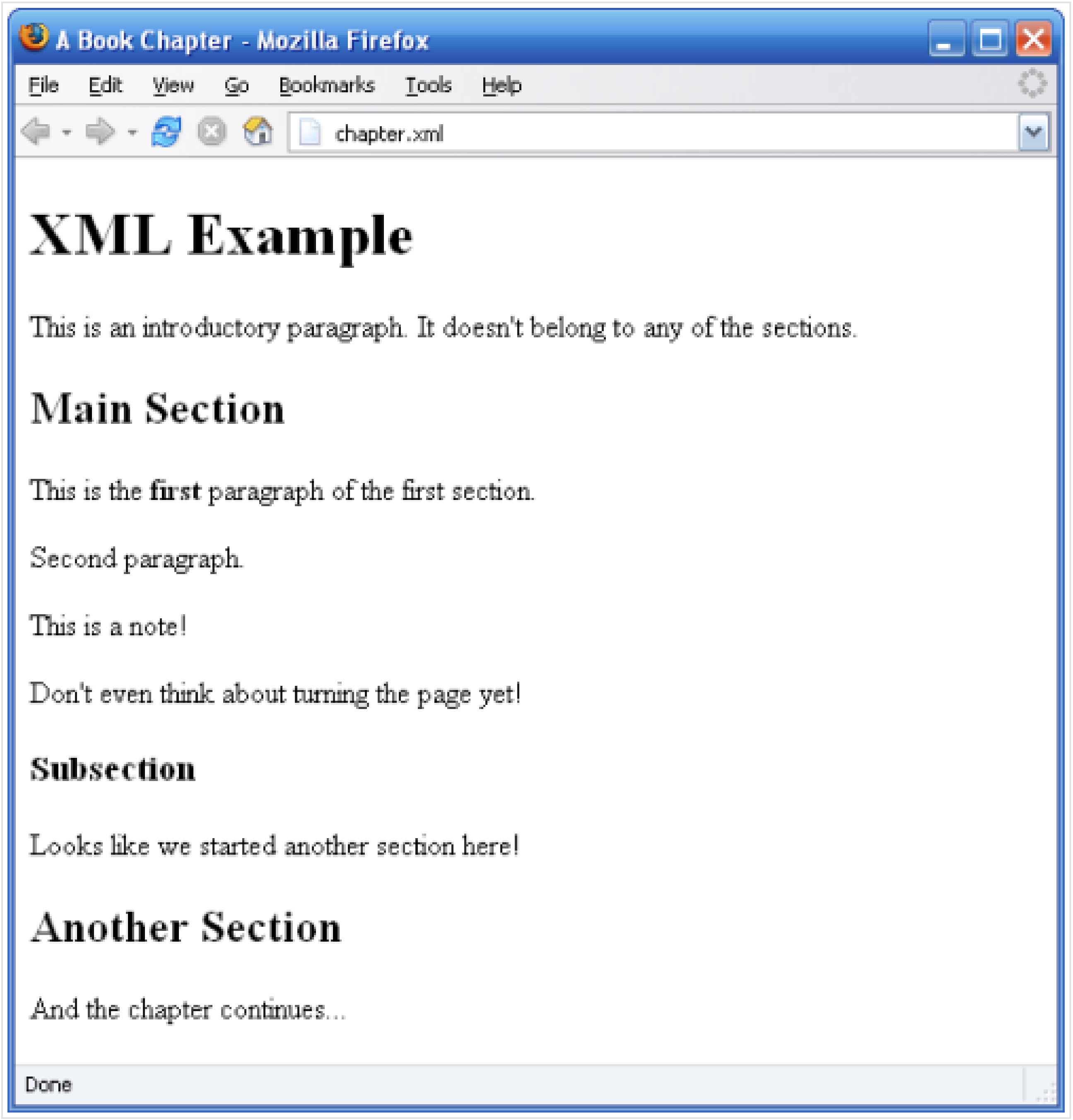
Example 4.5. `chapter2html.xsl` **(excerpt)**

```
<xsl:template match="chapter/title">

<h1><xsl:apply-templates/></h1>

</xsl:template>

<xsl:template match="chapter/section/title">

<h2><xsl:apply-templates/></h2>

</xsl:template>

<xsl:template match="chapter/section/section/title">

<h3><xsl:apply-templates/></h3>

</xsl:template>

<xsl:template match="chapter/section/section/section/title">
```

```
<h4><xsl:apply-templates/></h4>

</xsl:template>
```

Figure 4.2, “Viewing the chapter example with XPath. (Part 1)” shows how this code displays in the browser.

Figure 4.2. Viewing the chapter example with XPath. (Part 1)


A Book Chapter - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
chapter.xml
Done

XML Example

This is an introductory paragraph. It doesn't belong to any of the sections.

Main Section

This is the **first** paragraph of the first section.

Second paragraph.

This is a note!

Don't even think about turning the page yet!

Subsection

Looks like we started another section here!

Another Section

And the chapter continues...

View larger image.

We’re getting closer!

Matching Attribute Values with XPath

What about the paragraphs? Unlike the titles, they are not distinguishable by their placement in the document alone. Instead, the document uses the `type` attribute to distinguish normal paragraphs from introductions, tips, and warnings.

Luckily, XPath lets us specify matches based on attribute values, too. In XPath, we use a predicate (a condition in square brackets) to match an attribute value. To isolate `intro` paragraphs, for example, we would use the XPath expression `para[@type='intro']`.

We should definitely take advantage of this ability and distinguish each of our paragraph types visually. Let's italicize all introductory paragraphs, and put gray boxes around notes and warnings. We can also make sure that warnings are displayed in red text.

Now, we've already seen a template that can take care of normal paragraphs, which have no `type` attribute:

Example 4.6. `chapter2html.xsl` **(excerpt)**

```
<xsl:template match="para">

<p><xsl:apply-templates/></p>

</xsl:template>
```

Our template for introductory paragraphs is quite similar:

Example 4.7. `chapter2html.xsl` **(excerpt)**

```
<xsl:template match="para[@type='intro']" priority="1">

<p><i><xsl:apply-templates/></i></p>

</xsl:template>
```

Note the `priority` attribute on this template. Since an introductory paragraph would match both XPath expressions, `para` and `para[@type='intro']`, we need to give some indication as to which of the two templates should be used. By default, XSL templates have a priority between -0.5 and 0.5, depending on the

XPath expression in the `match` attribute. To make sure our introductory paragraphs will use this second template, we therefore assign a priority of `1`. Normal paragraphs will continue to use the first template, since they don't match the higher-priority second template.

With what we've just learned in mind, here are the templates for warnings and notes. Notice that we've added a `style` attribute to the opening `<p>` tag in each template to provide the desired style information for these paragraph types. In a practical application, you should instead put these style properties in a CSS file and `<link>` it to the HTML document. These templates would then use class attributes on the `<p>` tags to invoke the appropriate formatting.

Example 4.8. `chapter2html.xsl` **(excerpt)**

```
<xsl:template match="para[@type='warning']" priority="1">

<p style="background-color: #cccccc; border: thin solid;

width:300px; color:#ff0000;">

<xsl:apply-templates/>

</p>

</xsl:template>

<xsl:template match="para[@type='note']" priority="1">

<p style="background-color: #cccccc; border: thin solid;

width:300px;">

<b><xsl:apply-templates/></b>

</p>

</xsl:template>
```

Figure 4.3, "Viewing the chapter example with XPath. (Part 2)" shows the end result displayed in Firefox.

Using `value-of` *to Extract Information*

You'll notice the page title is the rather nondescript phrase, "A Book Chapter". How can we modify our template to display the actual chapter title in this spot instead?

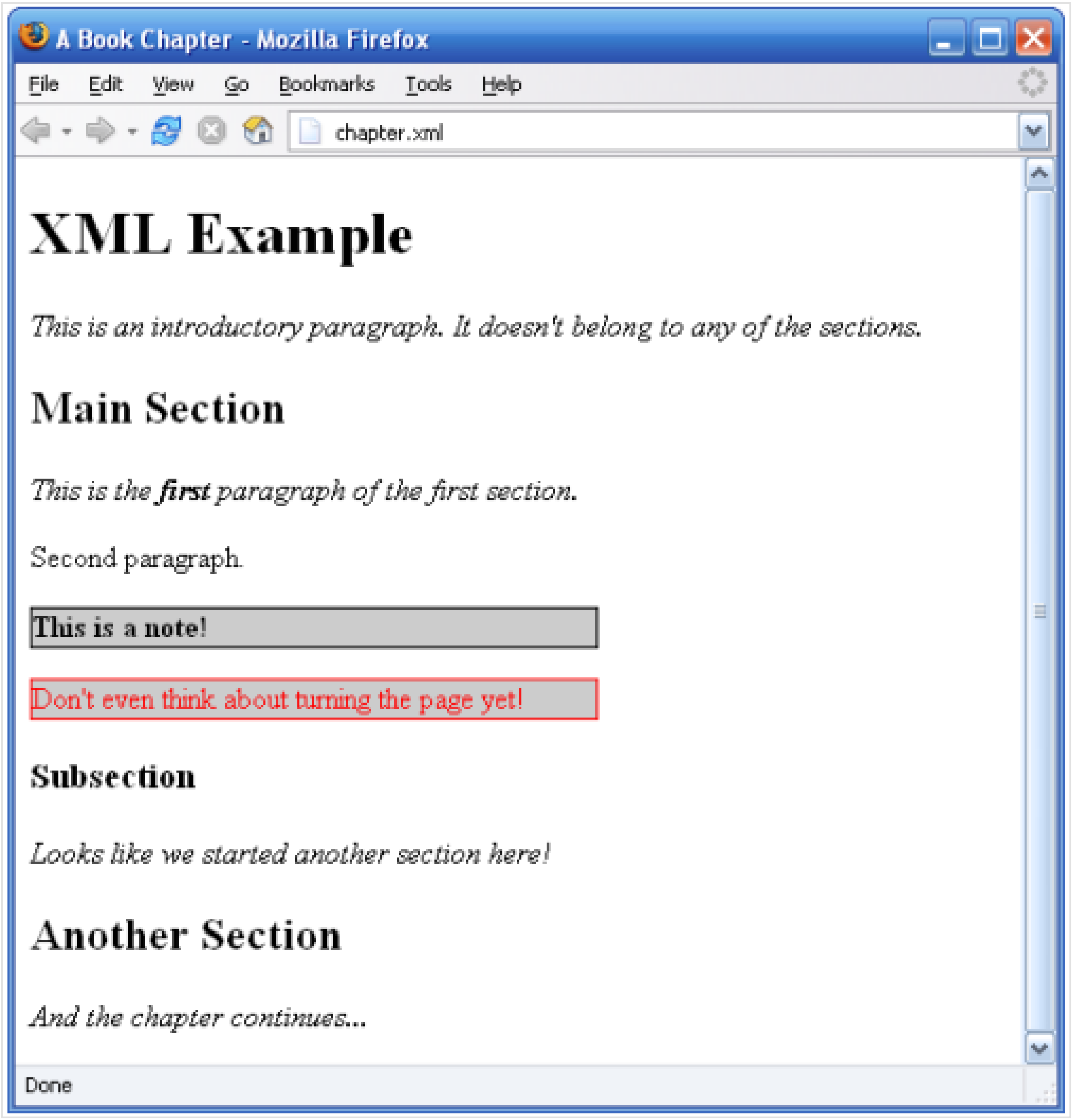
When you need to pull a simple piece of information out of the XML document without messing around with templates to process the element(s) that house it, you can use a `value-of` element to grab what you want with an XPath expression:

Example 4.9. `chapter2html.xsl` **(excerpt)**

```
<xsl:template match="/">

<html>

<head>

<title><xsl:value-of select="/chapter/title"/></title>

<meta http-equiv="content-type"

content="application/xhtml+xml; charset=iso-8859-1"/>

</head>

<body>

<xsl:apply-templates/>

</body>

</html>

</xsl:template>
```

As you can see, the `select` attribute is an XPath expression that searches for the value of the `title` within the `chapter`. With `value-of`, we can print that value out. Now our file displays something like the results shown in Figure 4.4, "Viewing the chapter example with XPath. (Part 3)". Notice the title bar of the browser window, which now contains the title of the chapter.

Figure 4.3. Viewing the chapter example with XPath. (Part 2)


A Book Chapter - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
chapter.xml
Done

XML Example

This is an introductory paragraph. It doesn't belong to any of the sections.

Main Section

*This is the **first** paragraph of the first section.*

Second paragraph.

This is a note!

Don't even think about turning the page yet!

Subsection

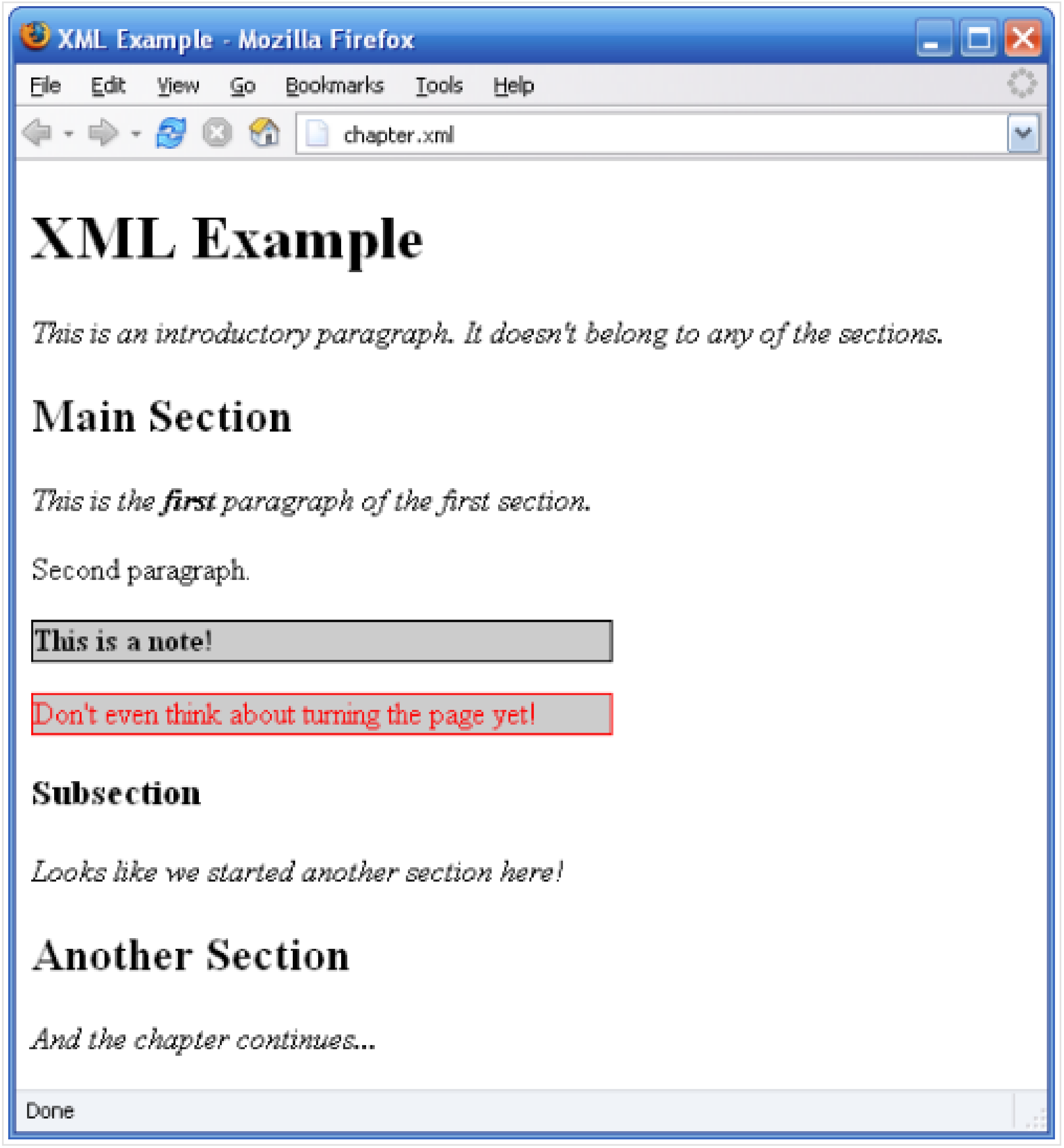
Looks like we started another section here!

Another Section

And the chapter continues...

View larger image.

Figure 4.4. Viewing the chapter example with XPath. (Part 3)



XML Example - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
chapter.xml

XML Example

This is an introductory paragraph. It doesn't belong to any of the sections.

Main Section

This is the ***first*** *paragraph of the first section.*

Second paragraph.

This is a note!

Don't even think about turning the page yet!

Subsection

Looks like we started another section here!

Another Section

And the chapter continues...

View larger image.

Viewing the chapter example with XPath. (Part 3)browsersview of XSLT styled XMLbook chapter example browser view

Our CMS Project

In the preceding chapters, we gathered requirements for our XML files, administration tool, and display components. In this chapter, I'd like to spend some time building the display pages for our project – the homepage, other internal pages, news sidebars, search widgets, and more.

Before we do that, though, let's recap the list of requirements we gathered for the display pages:

- The display side of our Website will only display articles and other content that has a status of "live."
- The search engine will retrieve articles by keywords, headlines, and descriptions, and only display those pieces that have a status of "live."
- The Website will display a list of authors by which site visitors can browse, but it only displays those authors who have live articles posted on the site.

Why Start with the Display Side?

You may be asking yourself, "Why is Tom starting with the display side? We haven't even built the admin tool for all the content it will display."

That's a good question. I decided to start with the display side because:

- It's much simpler than the admin tool, and gives us a chance to build some straightforward XML tools with PHP without having to get bogged down in detail.
- It means that we have to work from our requirements. Remember, we took the time to specify what each file would look like; now, all we have to do is work from these specs. As long as we continue to work from our specifications, everything will work together once it's done.

So, let's get started with our display pages. We'll begin with an include file that we can use on all of our pages.

Creating a Common Include File

Because our Website will entail some complex interaction between PHP and XML, it's a good idea to store your most needed functions and variables in a separate file, then include that file in all your other pages.

We're going to create this include file and start to add some information to it:

Example 4.10. `common.inc.php`

```
<?php
session_start();
```

```
$fileDir = $_SERVER['DOCUMENT_ROOT'] . '/xml/';
?>
```

This file will eventually contain many necessary variables that we'll use later in the project.

Before we go on to create a rudimentary homepage, let's create an include file that contains a search widget.

Creating a Search Widget Include File

All of our public display pages will offer a search widget, so it's a good idea to create a file that contains the needed form elements:

Example 4.11. `search.inc.php`

```
<form id="searchWidget" method="post" action="doSearch.php">
  Search site:
  <input name="term" type="text" id="term" />
  <input name="search" type="submit" id="search" value="Search" />
</form>
```

As with our common include file, we'll be using the PHP **include** command to include this form on all of our pages. In this case, we do so because it lowers maintenance costs: we only have to edit the form once to affect the whole site.

Notice that the action is set to a file called `doSearch.php`. We will work on that file soon – it's the file that will process XML and return search results to site visitors.

Building the Homepage

The most important page on the site is the homepage. That's where most of your visitors will likely begin, so you'll want to display as much information as you possibly can to interest them in going further.

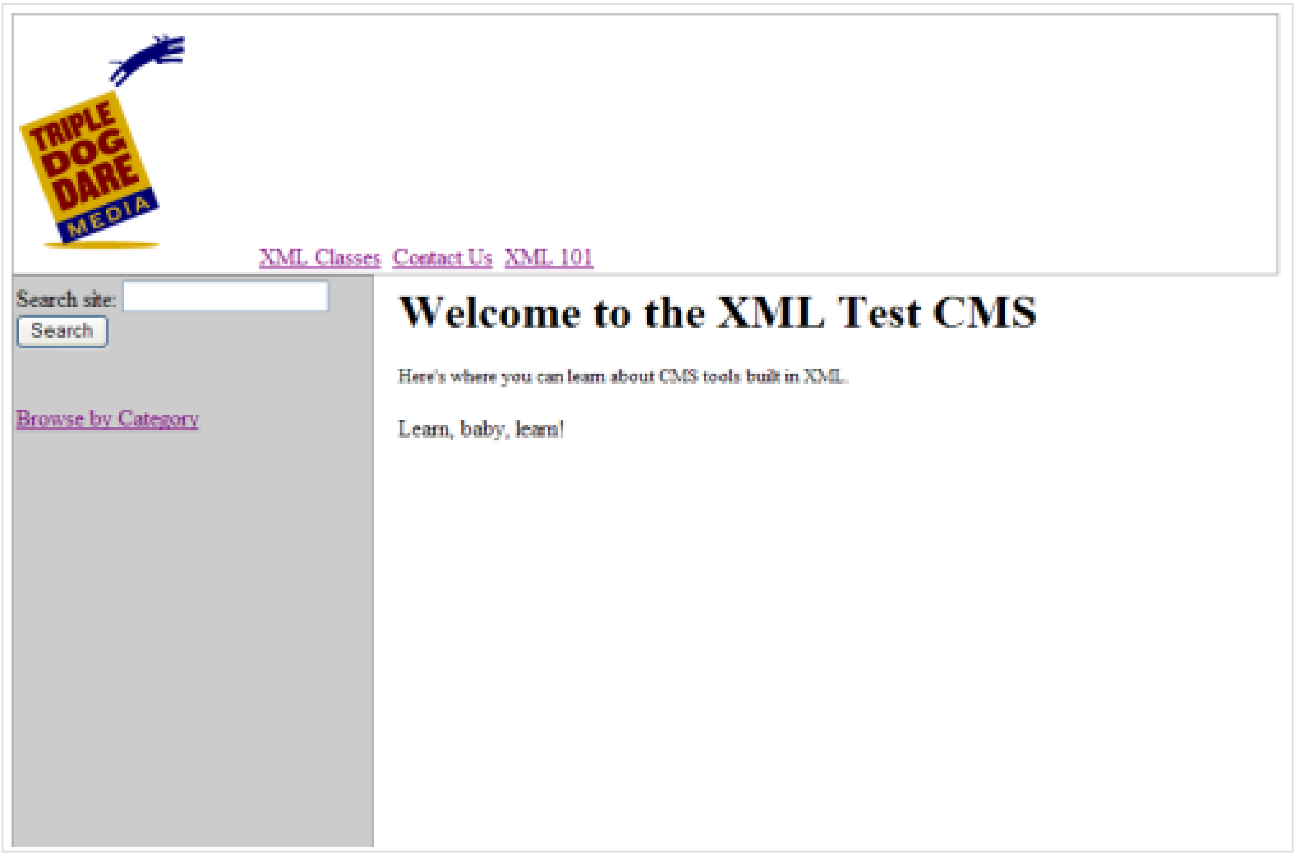
From a structural point of view, the pages of our site will consist of three `<div>` tags: a page header, a navigation menu, and the content area.

The header will hold global navigation elements. Like our search widget file, this navigation will be an include file – after all, we want to reuse these elements on other pages of the site.

For the homepage of our site, the navigation menu will contain our search widget and a list of current news items. In the main content area, we'll display our homepage copy along with links to articles and other content on the site.

We'll go through these sections one at a time. But, before we do, let's take a quick look at the appearance of our site's homepage – it's shown in Figure 4.5, "The appearance of the homepage.".

Figure 4.5. The appearance of the homepage.



Welcome to the XML Test CMS

Here's where you can learn about CMS tools built in XML.

Learn, baby, learn!

View larger image.

Building the Top Navigation Include File

Our top navigation will be placed in an include file. It will contain an image of the site's logo (hot-linked to the homepage for easy navigation), and a list of links that take users to each of the pages on the site.

This include file will make use of PHP 5's new SimpleXML functions. The great thing about the SimpleXML API is that it greatly simplifies the way you interact with, and extract information from, an XML document. Although a detailed look at SimpleXML will have to wait until ???, we'll cover the basics here.

Simply put, the `simplexml_load_file` function loads our entire XML document into a hierarchy of objects, which allows us to grab elements using PHP's familiar arrow notation. Imagine, for example, that you had this very simple XML document:

```
<person>
  <name>Tom</name>
  <age>33</age>
 </person>
```

After loading this XML document into a variable called `$person`, you would be able to examine the `name` element with `$person->name`. Likewise, you would be able to examine the `age` element with `$person->age`. If you're familiar with object oriented programming in PHP, you'll get the hang of it very quickly.

An even easier way to access XML elements with SimpleXML is to use an XPath query. You can pass a SimpleXML object just about any XPath statement, and it will retrieve the elements you need.

We'll get into a lot more detail later on, but for right now you can rest assured that at least one part of your job has been made easier!

Let's take a look at the code that will build the navigation bar at the top of the page. Then, we'll walk through it:

Example 4.12. `navtop.inc.php`

```
<div id="navTop">
 <a href="index.php"><img src="images/logo.gif" border="0"
    width="160" height="170" alt="Triple Dog Dare Media" /></a>
 <?php
 include_once 'common.inc.php';

 $handle = opendir($fileDir);
 while (($file = readdir($handle)) !== FALSE) {
  if (is_dir($ fileDir . $file)) continue;
  if (!eregi("^webcopy.*.xml$", $file)) continue;

  $webcopy = simplexml_load_file($fileDir . $file);
  if (count($webcopy->xpath('/webcopy[status="live"]'))) {
    $id = htmlentities($webcopy['id']);
    $label = htmlentities($webcopy->navigationlabel);
    echo "<a href="innerpage.php?id={$id}">{$label}</a> ";
  }
 }

 ?>
 </div>
```

Our first task is fairly simple: open the `xml` directory and find every XML file whose name begins with `webcopy`:

Example 4.13. `navtop.inc.php` **(excerpt)**

```
$handle = opendir($fileDir);
 while (($file = readdir($handle)) !== FALSE) {
  if (is_dir($fileDir . $file)) continue;
  if (!eregi("^webcopy.*.xml$", $file)) continue;
```

Remember, `$fileDir` is a variable set by `common.inc.php` to let this and other scripts on our site know where to find the XML files.

Regular Expressions

This code uses a *regular expression* to match the required file name pattern. For the lowdown on regular expressions in PHP, see Kevin Yank's book *Build Your Own Database Driven Website Using PHP & MySQL* (SitePoint), or refer to the PHP Manual.

With our Web copy XML files in hand, we'll load every such file using SimpleXML. Although this may seem like an expensive way to do things, you'll find that SimpleXML is extremely fast. We simply use the `simplexml_load_file` function to load the contents of each file into memory:

Example 4.14. `navtop.inc.php (excerpt)`

```
$webcopy = simplexml_load_file($fileDir . $file);
```

Once we have the desired file loaded into the `$webcopy` variable, we can start to look at the XML document it contains. In this case, we're only interested in the files whose status is "live," so we use SimpleXML to check that the `status` element does indeed contain a text value of `live`:

Example 4.15. `navtop.inc.php` **(excerpt)**

```
if (count($webcopy->xpath('/webcopy[status="live"]'))) {
```

Here, we're using SimpleXML's `xpath` method to check if the `webcopy` element at the root of the document contains a `status` element with a value of `live`. The method returns an array of elements that match the criteria specified; in this case that array will either contain a reference to the `webcopy` element in the file (if the `status` is `live`), or it will be empty. We use PHP's `count` function to check.

If the file passes the test, we pull out the value of the `webcopy` element's `id` attribute and the value contained in the nested `navigationlabel` element.

Example 4.16. `navtop.inc.php` **(excerpt)**

```
$id = htmlentities($webcopy['id']);
```

$label = htmlentities($webcopy->navigationlabel);

As you can see, attributes are referenced as elements in an array (`$webcopy['id']`), while nested elements are referenced as object properties (`$webcopy->navigationlabel`).

With these values in hand, we can print out appropriate links for our page navigation:

Example 4.17. `navtop.inc.php` **(excerpt)**

```
echo "<a href="webcopy.php?id={$id}">{$label}</a>
```

Let's move on to the rest of the homepage.

Building the Bottom Half of the Homepage

Remember when I said that our homepage would be made up of three `<div>` tags? Well, we've just taken care of the first – the page header. Let's now talk about the remaining two `divs` that sit beneath the first.

The file for our homepage will be called `index.php`. This file includes both the `common.inc.php` and `navtop.inc.php` files as needed. It then goes on to produce the secondary navigation and content `divs` (`navSide` and `mainContent`, respectively).

Example 4.18. `index.php`

```
<?php
include_once 'common.inc.php';
$file = $fileDir . 'homepage.xml';
$homePage = simplexml_load_file($file);
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
 <title><?php echo htmlentities($homePage->headline); ?></title>
 <meta http-equiv="Content-Type"
     content="text/html; charset=iso-8859-1" />
 <link rel="stylesheet" href="xmlcms.css" type="text/css" />
</head>
<body>
<?php
include 'navtop.inc.php';
?>
<div id="navSide">
 <?php
 include 'search.inc.php';
 include 'news.inc.php';
 ?>
</div>
<div id="mainContent">
 <?php
 echo '<h1>' . htmlentities($homePage->headline) . '</h1>';
 echo '<p><small>' . htmlentities($homePage->description) .
     '</small></p>';
 echo $homePage->body;
 ?>
</div>
</body>
</html>
```

It looks really simple, doesn’t it? In this file, we’re using a variety of includes and PHP functions to do a lot of the dirty work for us. We’ll also use this approach when we want to build the other display pages for articles, Web copy, and the like.

The only part that is somewhat complicated is the first few lines:

Example 4.19. `index.php` **(excerpt)**

```
<?php
include_once 'common.inc.php';
$file = $fileDir . 'homepage.xml';
$homePage = simplexml_load_file($file);
?>
...
<title><?php echo htmlentities((string)$homePage->headline);
 ?></title>
```

In this code, we open the file called `homepage.xml` in the `xml` directory, and then **echo** out the contents of the `headline` element as the page title.

For the left-side navigation div, we will use two includes:

Example 4.20. `index.php` (excerpt)

```
<div id="navSide">
  <?php
  include 'search.inc.php';
  include 'news.inc.php';
  ?>
</div>
```

The first include is the search widget that we built earlier on. The second should produce a listing of live news items, but we haven't built that yet.

For the most part, our news include file will be very similar in structure to the code we used in `navtop.inc.php`. All we're doing is extracting news items that have a `status` of `live`:

Example 4.21. `news.inc.php` (excerpt)

```
<?php
include_once 'common.inc.php';

$handle = opendir($fileDir);
echo '<p>';
while (($file = readdir($handle)) !== FALSE) {
  if (is_dir($fileDir . $file)) continue;
  if (!eregi('^news.*.xml$', $file)) continue;

  $news = simplexml_load_file($fileDir . $file);
  if (count($news->xpath('/news[status="live"]'))) {
    $id = htmlentities($news['id']);
    $label = htmlentities($news->headline);
    echo "<a href="innerpage.php?id={$id}">{$label}</a><br />";
  }
 }
 echo '</p>';

 ?>
```

Now that we've completed the left side of the homepage, it's time to pull together the right side of the page. This area will display the headline and body copy that's stored for the homepage in a file called `homepage.xml`. Since we've already loaded this file to obtain the page title, we can continue using the `$homePage` variable to pull out the values we need:

Example 4.22. `index.php` **(excerpt)**

```
<div id="mainContent">
 <?php
 echo '<h1>' . htmlentities($homePage->headline) . '</h1>';
 echo '<p><small>' . htmlentities($homePage->description) .
     '</small></p>';
 echo $homePage->body;
 ?>
</div>
</body>
</html>
```

Writing the Style Sheet

This isn't a book about CSS page layout, so I won't dwell on the details of the site's style sheet. For the sake of completeness, however, here's the code, which ensures our pages are laid out the way we intended:

Example 4.23. `xmlcms.css`

```
body {
 color: #000;
 background: #fff;
 font-family: Helvetica, Arial, sans-serif;
 margin: 0;
 padding: 0;
}
#navTop {
 margin: 12px 12px 0 12px;
```

```
  border: 1px solid #999;
  padding: 2px;
 }
 #navSide {
  position: absolute;
  width: 250px;
  min-height: 400px;
  left: 12px;
  background-color: #ccc;
  border: 1px solid #999;
  margin-top: -1px;
  padding: 2px;
 }
 #mainContent {
  margin: 8px 8px 8px 280px;
 }
```

Creating an Inner Page

We have the homepage all roughed out. Now, we need to build another template that will handle the display of the rest of the site's content. We'll get this work started now, and come back to it later as necessary.

For now, all we have to do is make a copy of `index.php` and call it `innerpage.php` – this will maintain the same includes and layout as our homepage. We'll make a few minor changes to this new template, in particular, to the code that is used to extract information from the correct file in the `xml` directory.

An `id` variable will be passed in the query string, which will correspond to the filename of the XML file that contains the associated content. So the ID `webcopy3` will correspond to a file named `webcopy3.xml` in the `xml` directory.

Since we're using input from the browser (the `id` variable) as a filename in our script, we must be sure to check that the value passed is not a security risk. Otherwise, we could find our script turned against us as a clever hacker submits a value that points to some sensitive file on the system. For our purposes, a regular expression that verifies that the variable contains an alphanumeric string (only numbers and letters) will suffice.

With these considerations in mind, here's the code that loads the XML file associated with the supplied ID:

Example 4.24. `innerpage.php` **(excerpt)**

```
<?php
include_once 'common.inc.php';
if (!isset($_GET['id']) or !eregi('^[a-z0-9]+$', $_GET['id']))
  return;
$file = $fileDir . $_GET['id'] . '.xml';
$inner = simplexml_load_file($file);
?>
```

With the file loaded, we must pull out the values inside for display in the template. In this instance, we're using a single template file to display two different types of content: news items (`news123.xml`) and Web copy (`webcopy123.xml`). If you refer back to Chapter 2, *XML in Practice*, where we defined these XML formats, you'll see that the Web copy has `navigationlabel` and `body` elements that news items do not. We'll have to detect these to make sure our template displays the right thing.

The best way to do this with the SimpleXML API is to use an XPath query. For example, we want to use the `navigationlabel` element for the page title, but if no such element exists we want to fall back on the `headline` element. Here's the code:

Example 4.25. `innerpage.php` **(excerpt)**

```
<title>
<?php
if (count($inner->xpath('navigationlabel'))) {
  echo htmlentities($inner->navigationlabel);
} elseif (count($inner->xpath('headline'))) {
  echo htmlentities($inner->headline);
}
?>
</title>
```

With all this in mind, you should be in a position to understand the complete template at a glance.

Example 4.26. `innerpage.php`

```
<?php
include_once 'common.inc.php';
if (!isset($_GET['id']) or !eregi('^[a-z0-9]+$', $_GET['id']))
```

```
 return;
$file = $fileDir . $_GET['id'] . '.xml';
$inner = simplexml_load_file($file);
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>
<?php
if (count($inner->xpath('navigationlabel'))) {
 echo htmlentities($inner->navigationlabel);
} elseif (count($inner->xpath('headline'))) {
 echo htmlentities($inner->headline);
}
?>
</title>
<meta http-equiv="content-type"
   content="text/html; charset=iso-8859-1" />
<link rel="stylesheet" href="xmlcms.css" type="text/css" />
</head>
<body>
<?php
include 'navtop.inc.php';
?>
<div id="navSide">
 <?php
 include 'search.inc.php';
 include 'news.inc.php';
 ?>
</div>
<div id="mainContent">
 <?php
 echo '<h1>' . htmlentities($inner->headline) . '</h1>';
 echo '<p><small>' . htmlentities($inner->description) .
     '</small></p>';
 if (count($inner->xpath('body'))) {
   echo $inner->body;
 }
 ?>
</div>
</body>
</html>
```

That's really all we need at the moment – we have the foundations of a Website working already! We don't have much formatting yet, nor a working search engine, but the display side is coming together quite nicely.

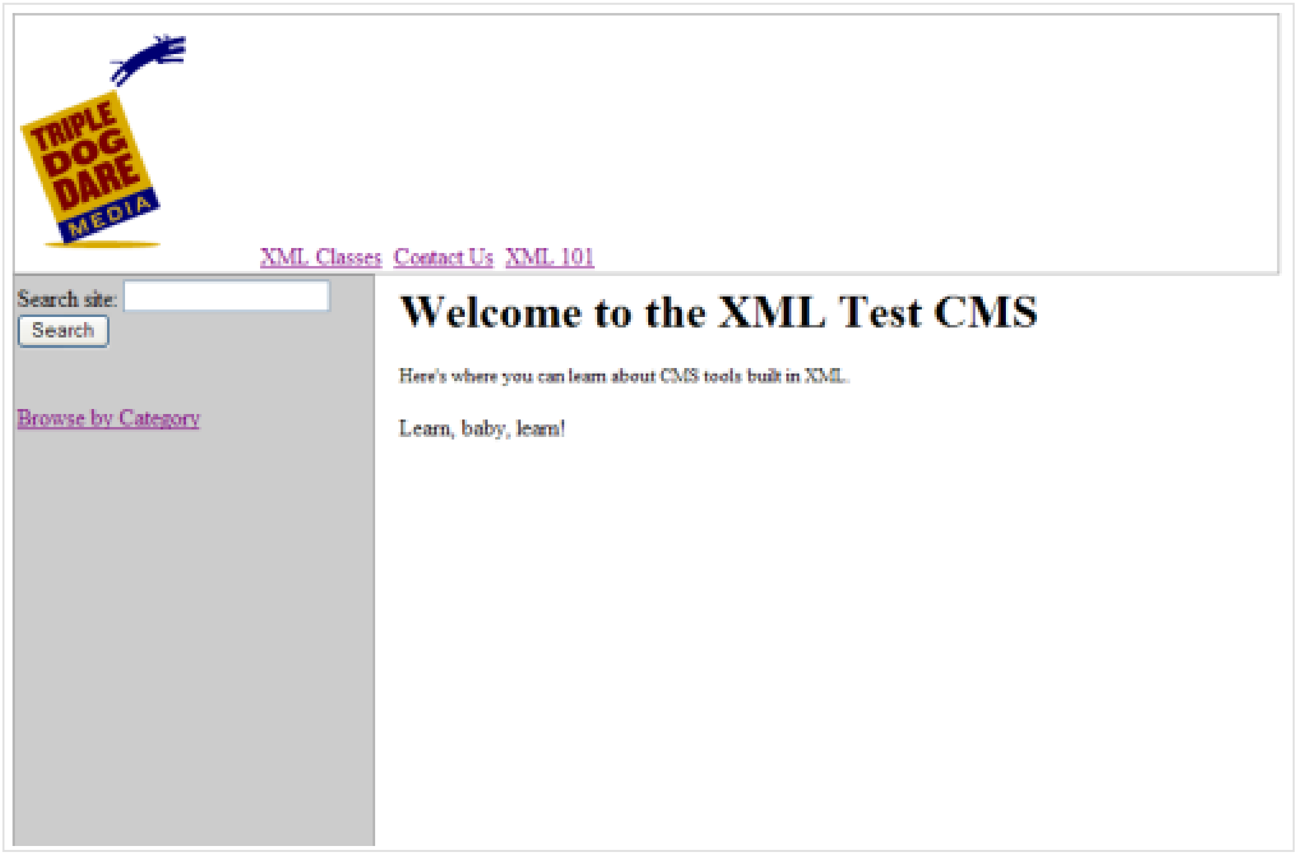
What does our sample site look like so far? Well, since we haven't created any XML documents yet, yours might not work at all. On my system, however, I've inserted a number of files, which I've supplied for you in the code archive for this chapter, and the site looks like that shown in Figure 4.6, "Displaying the CMS project so far.".

Over the next few chapters, we'll create XML documents with an administration tool, and the project will really start to come together.

Summary

In this chapter, we got a closer look at XSLT as we roughed out the display pages we'll need for our project. In ???, we'll look even more closely at XSLT, as we learn some of the more programmatic aspects of the language, such as loops, variables, and branches. We'll also fill in the elements we'll need for the display side, such as a working search engine, some formatting rules, and other details.

Figure 4.6. Displaying the CMS project so far.



Welcome to the XML Test CMS

Here's where you can learn about CMS tools built in XML.

Learn, baby, learn!

View larger image.

That's it for this excerpt of "No Nonsense XML Web Development with PHP! What's next?

Download these chapters in PDF format, and you'll have a copy you can refer to at any time.

Review the book's table of contents to find out exactly what's included.

Buy your own copy of the book now, right here at SitePoint.com.

We hope you enjoy No Nonsense XML Web Development with PHP.
