Chapter 8

Working with XML

Key Skills & Concepts

- Understand basic XML concepts and technologies
- Understand PHP's SimpleXML and DOM extensions
- Access and manipulate XML documents with PHP
- Create new XML documents from scratch using PHP
- Integrate third-party RSS feeds in a PHP application
- Convert between SQL and XML using PHP

The Extensible Markup Language (XML) is a widely accepted standard for data description and exchange. It allows content authors to "mark up" their data with customized machine-readable tags, thereby making data easier to classify and search. XML also helps enforce a formal structure on content, and it provides a portable format that can be used to easily exchange information between different systems.

PHP has included support for XML since PHP 4, but it was only in PHP 5 that the various XML extensions in PHP were standardized to use a common XML parsing toolkit. This chapter introduces you to two of PHP's most useful and powerful XML processing extensions—SimpleXML and DOM—and includes numerous code examples and practical illustrations of using XML in combination with PHP-based applications.

Introducing XML

Before getting into the nitty-gritty of manipulating XML files with PHP, it's instructive to spend some time getting familiar with XML. If you're new to XML, the following section provides a grounding in basic XML, including an overview of XML concepts and technologies. This information will be helpful to understand the more advanced material in subsequent sections.

XML Basics

Let's begin with a very basic question: what is XML, and why is it useful?

XML is a language that helps document authors describe the data in a document, by "marking it up" with custom tags. These tags don't come from a predefined list; instead,

XML encourages authors to create their own tags and structure, suited to their own particular requirements, as a way to increase flexibility and usability. This fact, coupled with the Recommendation status bestowed on it by the W3C in 1998, has served to make XML one of the most popular ways to describe and store structured information on (and off) the Web.

XML data is physically stored in text files. This makes XML documents very portable, because every computer system can read and process text files. Not only does this facilitate data sharing, but it also allows XML to be used in a wide variety of applications. For example, the Rich Site Summaries (RSS) and Atom Weblog feed formats are both based on XML, as is Asynchronous JavaScript and XML (AJAX) and the Simple Object Access Protocol (SOAP).

Ask the Expert

Q: What programs can I use to create or view an XML file?

A: On a UNIX/Linux system, both `vi` and `emacs` can be used to create XML documents, while Notepad remains a favorite on Windows systems. Both Microsoft Internet Explorer and Mozilla Firefox have built-in XML support and can read and display an XML document in a hierarchical tree view.

Anatomy of an XML Document

Internally, an XML document is made up of various components, each one serving a specific purpose. To understand these components, consider the following XML document, which contains a recipe for spaghetti bolognese:

```
1.  <?xml version='1.0'?>
2.  <recipe>
3.    <ingredients>
4.      <item quantity="250" units="gm">Beef mince</item>
5.      <item quantity="200" units="gm">Onions</item>
6.      <item quantity="75" units="ml">Red wine</item>
7.      <item quantity="12">Tomatoes</item>
8.      <item quantity="2" units="tbsp">Parmesan cheese</item>
9.      <item quantity="200" units="gm">Spaghetti</item>
10.   </ingredients>
11.   <method>
12.     <step number="1">Chop and fry the onions.</step>
13.     <step number="2">Add the mince to the fried onions &amp;
continue frying.</step>
```

```
14.     <step number="3">Puree the tomatoes and blend them into the
mixture with the wine.</step>
15.     <step number="4">Simmer for an hour.</step>
16.     <step number="5">Serve on top of cooked pasta with Parmesan
cheese.</step>
17.   </method>
18. </recipe>
```

This XML document contains a recipe, broken up into different sections; each section is further "marked up" with descriptive tags to precisely identify the type of data contained within it. Let's look at each of these in detail.

1. Every XML document must begin with a declaration that states the version of XML being used; this declaration is referred to as the *document prolog,* and it can be seen in Line 1 of the preceding XML document. In addition to the version number, this document prolog may also contain character encoding information and Document Type Definition (DTD) references (for data validation).
2. The document prolog is followed by a nested series of *elements* (Lines 2–18). Elements are the basic units of XML; they typically consist of a pair of opening and closing tags that enclose some textual content. Element names are user-defined; they should be chosen with care, as they are intended to describe the content sandwiched between them. Element names are case-sensitive and must begin with a letter, optionally followed by more letters and numbers. The outermost element in an XML document—in this example, the element named `<recipe>` on Line 2—is known as the *document element* or the *root element.*
3. The textual data enclosed within elements is known, in XML parlance, as *character data.* This character data can consist of strings, numbers, and special characters (with some exceptions: angle brackets and ampersands within textual data should be replaced with the entities `<`, `$gt;`, and `&` respectively to avoid confusing the XML parser when it reads the document). Line 13, for example, uses the `&` entity to represent an ampersand within its character data.
4. Finally, elements can also contain *attributes,* which are name-value pairs that hold additional information about the element. Attribute names are case-sensitive and follow the same rules as element names. The same attribute name cannot be used more than once within the same element, and attribute values should always be enclosed in quotation marks. Lines 4–9 and 12–16 in the example document illustrate the use of attributes to provide additional descriptive information about the element to which they are attached; for example, the `'units'` attribute specifies the unit measure for each ingredient.

XML documents can also contain various other components: namespaces, processing instructions, and CDATA blocks. These are a little more complex, and you won't see them in any of the examples used in this chapter; however, if you're interested in learning more about them, take a look at the links at the end of the chapter for more detailed information and examples.

Ask the Expert

Q: Can I create elements that don't contain anything?

A: Sure. The XML specification supports elements that hold no content and therefore do not require a closing tag. To close these elements, simply add a slash to the end of the opening tag, as in the following code snippet:

```
The line breaks <br /> here.
```

Well-Formed and Valid XML

The XML specification makes an important distinction between well-formed and valid documents.

- A *well-formed document* follows all the rules for element and attribute names, contains all essential declarations, contains one (and only one) root element, and follows a correctly nested element hierarchy below the root element. All the XML documents you'll see in this chapter are well-formed documents.
- A *valid document* is one which meets all the conditions of being well-formed and also conforms to additional rules laid out in a Document Type Definition (DTD) or XML Schema. This chapter doesn't discuss DTDs or XML Schemas in detail, so you won't see any examples of this type of document; however, you'll find many examples of such documents online, and in the resource links at the end of the chapter.

XML Parsing Methods

Typically, an XML document is processed by a software application known as an XML parser. This parser reads the XML document using one of two approaches, the Simple API for XML (SAX) approach or the Document Object Model (DOM) approach:

- A SAX parser works by traversing an XML document sequentially, from beginning to end, and calling specific user-defined functions as it encounters different types of XML constructs. Thus, for example, a SAX parser might be programmed to call one

function to process an attribute, another one to process a starting element, and yet a third one to process character data. The functions called in this manner are responsible for actually processing the XML construct found, and any information stored within it.

- A DOM parser, on the other hand, works by reading the entire XML document in one fell swoop and converting it to a hierarchical "tree" structure in memory. The parser can then be programmed to traverse the tree, jumping between "sibling" or "child" branches of the tree to access specific pieces of information.

Each of these methods has pros and cons: SAX reads XML data in "chunks" and is efficient for large files, but it requires the programmer to create customized functions to handle the different elements in an XML file. DOM requires less customization but can rapidly consume memory for its actions and so is often unsuitable for large XML data files. The choice of method thus depends heavily on the requirements of the application in question.

XML Technologies

As XML's popularity has increased, so too has the list of technologies that use it. Here are a few that you might have heard about already:

- **XML Schema** XML Schemas define the structure and format of XML documents, allowing for more flexible validation and support for datatyping, inheritance, grouping, and database linkage.
- **XLink** XLink is a specification for linking XML data structures together. It allows for more sophisticated link types than regular HTML hyperlinks, including links with multiple targets.
- **XPointer** XPointer is a specification for navigating the hierarchical tree structure of an XML document, easily finding elements, attributes, and other data structures within the document.
- **XSL** The Extensible Stylesheet Language (XSL) applies formatting rules to XML documents and "transforms" them from one format to another.
- **XHTML** XHTML combines the precision of XML markup with the easy-to-understand tags of HTML to create a newer, more standards-compliant version of HTML.
- **XForms** XForms separates the information gathered in a Web form from the form's appearance, allowing for more stringent validation and easier reuse of forms in different media.

- **XML Query** XML Query allows developers to query XML document and generate result sets, in much the same way that SQL is used to query and retrieve database records.
- **XML Encryption and XML Signature** XML Encryption is a means of encrypting and decrypting XML documents, and representing the resulting data. It is closely related to XML Signature, which provides a way to represent and verify digital signatures with XML.
- **SVG** Scalable Vector Graphics (SVG) uses XML to describe vector or raster graphical images, with support for alpha masks, filters, paths, and transformations.
- **MathML** MathML uses XML to describe mathematical expressions and formulae, such that they can be easily rendered by Web browsers.
- **SOAP** The Simple Object Access Protocol (SOAP) uses XML to encode requests and responses between network hosts using HTTP.

Ask the Expert

Q: When should I use an attribute, and when should I use an element?

A: Both attributes and elements contain descriptive data, so it's often a matter of judgment as to whether a particular piece of information is better stored as an element or as an attribute. In most cases, if the information is hierarchically structured, elements are more appropriate containers; on the other hand, attributes are better for information that is ancillary or does not lend itself to a formal structure.

For a more formal discussion of this topic, take a look at the IBM developerWorks article at www.ibm.com/developerworks/xml/library/x-eleatt.html, which discusses the issues involved in greater detail.

Try This 8-1 Creating an XML Document

Now that you know the basics of XML, let's put that knowledge to the test by creating a well-formed XML document and viewing it in a Web browser. This document will describe a library of books using XML. Each entry in the document will contain information on a book's title, author, genre, and page count.

(continued)

To create this XML document, pop open your favorite text editor and enter the following markup (*library.xml*):

```
<?xml version="1.0"?>
<library>
  <book id="1" genre="horror" rating="5">
    <title>The Shining</title>
    <author>Stephen King</author>
    <pages>673</pages>
  </book>
  <book id="2" genre="suspense" rating="4">
    <title>Shutter Island</title>
    <author>Dennis Lehane</author>
    <pages>390</pages>
  </book>
  <book id="3" genre="fantasy" rating="5">
    <title>The Lord Of The Rings</title>
    <author>J. R. R. Tolkien</author>
    <pages>3489</pages>
  </book>
  <book id="4" genre="suspense" rating="3">
    <title>Double Cross</title>
    <author>James Patterson</author>
    <pages>308</pages>
  </book>
  <book id="5" genre="horror" rating="4">
    <title>Ghost Story</title>
    <author>Peter Straub</author>
    <pages>389</pages>
  </book>
  <book id="6" genre="fantasy" rating="3">
    <title>Glory Road</title>
    <author>Robert Heinlein</author>
    <pages>489</pages>
  </book>
  <book id="7" genre="horror" rating="3">
    <title>The Exorcist</title>
    <author>William Blatty</author>
    <pages>301</pages>
  </book>
  <book id="8" genre="suspense" rating="2">
    <title>The Camel Club</title>
    <author>David Baldacci</author>
    <pages>403</pages>
  </book>
</library>
```

Save this file to a location under your Web server's document root, and name it *library.xml.* Then, start up your Web browser, and browse to the URL corresponding to the file location. You should see something like Figure 8-1.

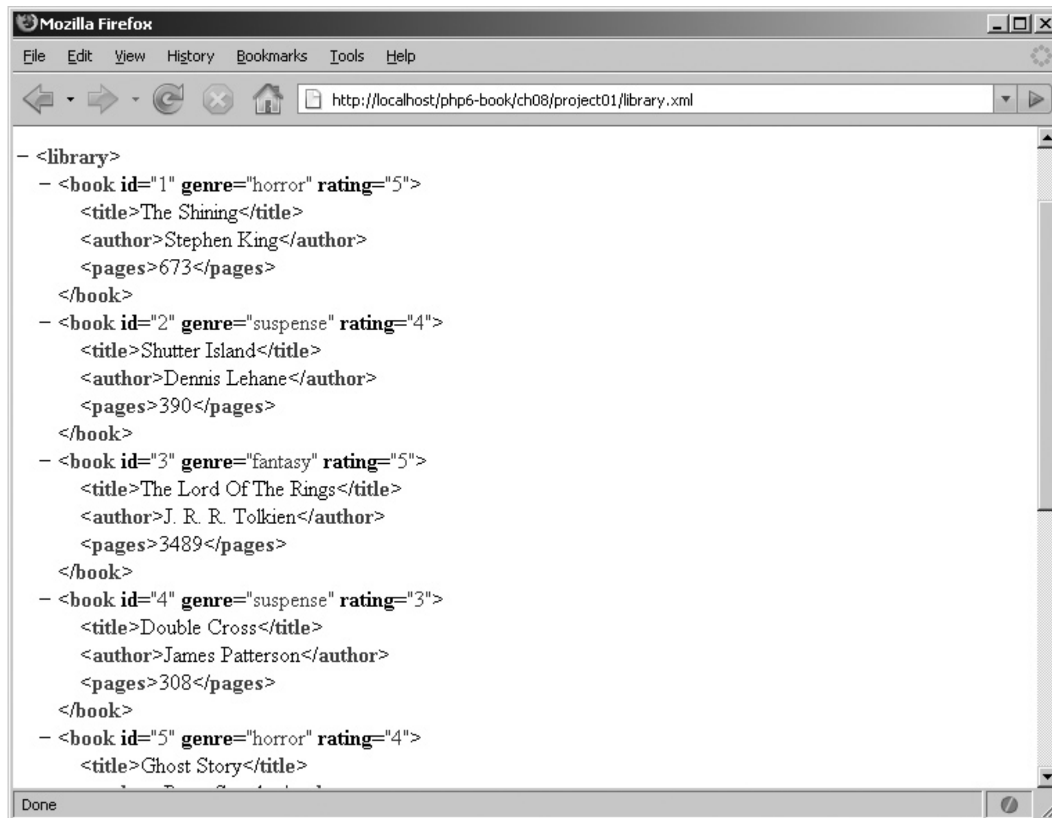

Figure 8-1 An XML document, as seen in Mozilla Firefox

Using PHP's SimpleXML Extension

Although PHP has support for both DOM and SAX parsing methods, by far the easiest way to work with XML data in PHP is via its SimpleXML extension. This extension, which is enabled by default in PHP 5, provides a user-friendly and intuitive interface to reading and processing XML data in a PHP application.

Working with Elements

SimpleXML represents every XML document as an object and turns the elements within it into a hierarchical set of objects and object properties. Accessing a particular element now becomes as simple as using `parent->child` notation to traverse the object tree until that element is reached.

To illustrate how this works in practice, consider the following XML file (*address.xml*):

```
<?xml version='1.0'?>
<address>
  <street>13 High Street</street>
  <county>Oxfordshire</county>
  <city>
    <name>Oxford</name>
    <zip>OX1 1BA</zip>
  </city>
  <country>UK</country>
</address>
```

Here's a PHP script that uses SimpleXML to read this file and retrieve the city name and ZIP code:

```
<?php
// load XML file
$xml = simplexml_load_file('address.xml') or die ("Unable to load XML!");

// access XML data
// output: 'City: Oxford \n Postal code: OX1 1BA\n'
echo "City: " . $xml->city->name . "\n";
echo "Postal code: " . $xml->city->zip . "\n";
?>
```

To read an XML file with SimpleXML, use the `simplexml_load_file()` function and pass it the disk path to the target XML file. This function will then read and parse the XML file and, assuming it is well-formed, return a SimpleXML object representing the document's root element. This object is only the top level of a hierarchical object tree that mirrors the internal structure of the XML data: elements below the root element are represented as properties or child objects and can thus be accessed using the standard `parentObject->childObject` notation.

TIP

If your XML data is in a string variable instead of a file, use the `simplexml_load_string()` function to read it into a SimpleXML object.

Multiple instances of the same element at the same level of the XML document tree are represented as arrays. These can easily be processed using PHP's loop constructs. To illustrate, consider this next example, which reads the *library.xml* file developed in the preceding section and prints the title and author names found within it:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");
```

```
// loop over XML data as array
// print book titles and authors
// output: 'The Shining is written by Stephen King. \n ...'
foreach ($xml->book as $book) {
  echo $book->title . "  is written by " . $book->author . ".\n";
}
?>
```

Here, a `foreach` loop iterates over the `<book>` objects generated from the XML data, printing each object's `'title'` and `'author'` properties.

You can also count the number of elements at particular level in the XML document with a simple call to `count()`. The next listing illustrates, counting the number of `<book>`s in the XML document:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");

// loop over XML data as array
// print count of books
// output: '8 book(s) found.'
echo count($xml->book) . ' book(s) found.';
?>
```

Working with Attributes

If an XML element contains attributes, SimpleXML has an easy way to get to these as well: attributes and values are converted into keys and values of a PHP associative array and can be accessed like regular array elements.

To illustrate, consider the following example, which reads the *library.xml* file from the preceding section and prints each book title found, together with its `'genre'` and `'rating'`:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");

// access XML data
// for each book
// retrieve and print 'genre' and 'rating' attributes
// output: 'The Shining \n Genre: horror \n Rating: 5 \n\n ...'
foreach ($xml->book as $book) {
  echo $book->title . "\n";
  echo "Genre: " . $book['genre'] . "\n";
  echo "Rating: " . $book['rating'] . "\n\n";
}
?>
```

In this example, a `foreach` loop iterates over the `<book>` elements in the XML data, turning each into an object. Attributes of the book element are represented as elements of an associative array and can thus be accessed by key: the key `'genre'` returns the value of the `'genre'` attribute, and the key `'rating'` returns the value of the `'rating'` attribute.

Try This 8-2 Converting XML to SQL

Now that you know how to read XML elements and attributes, let's look at a practical example of SimpleXML in action. This next program reads an XML file and converts the data within it into a series of SQL statements, which can be used to transfer the data into a MySQL or other SQL-compliant database.

Here's the sample XML file (*inventory.xml*):

```
<?xml version='1.0'?>
<items>
  <item sku="123">
    <name>Cheddar cheese</name>
    <price>3.99</price>
  </item>
  <item sku="124">
    <name>Blue cheese</name>
    <price>5.49</price>
  </item>
  <item sku="125">
    <name>Smoked bacon (pack of 6 rashers)</name>
    <price>1.99</price>
  </item>
  <item sku="126">
    <name>Smoked bacon (pack of 12 rashers)</name>
    <price>2.49</price>
  </item>
  <item sku="127">
    <name>Goose liver pate</name>
    <price>7.99</price>
  </item>
  <item sku="128">
    <name>Duck liver pate</name>
    <price>6.49</price>
  </item>
</items>
```

And here's the PHP code that converts this XML data into SQL statements (*xml2sql.php*):

```
<?php
// load XML file
$xml = simplexml_load_file('inventory.xml') or die ("Unable to load XML!");

// loop over XML <item> elements
// access child nodes and interpolate with SQL statement
foreach ($xml as $item) {
  echo "INSERT INTO items (sku, name, price) VALUES ('" . addslashes($item['sku']) .
"','" . addslashes($item->name) . "','" . addslashes($item->price) . "');\n";
}
?>
```

This script should be simple to understand if you've been following along: it iterates over all the `<item>` elements in the XML document, using `object->property` notation to access each item's `<name>` and `<price>`. The `'sku'` attribute of each `<item>` is similarly accessed via the `'sku'` key of each item's attribute array. The values retrieved in this fashion are then interpolated into an SQL `INSERT` statement.

This statement would normally then be supplied to a function such as `mysql_query()` or `sqlite_query()` for insertion into a MySQL or SQLite database; for purposes of this example, it's simply printed to the output device.

Figure 8-2 illustrates the output of this script.

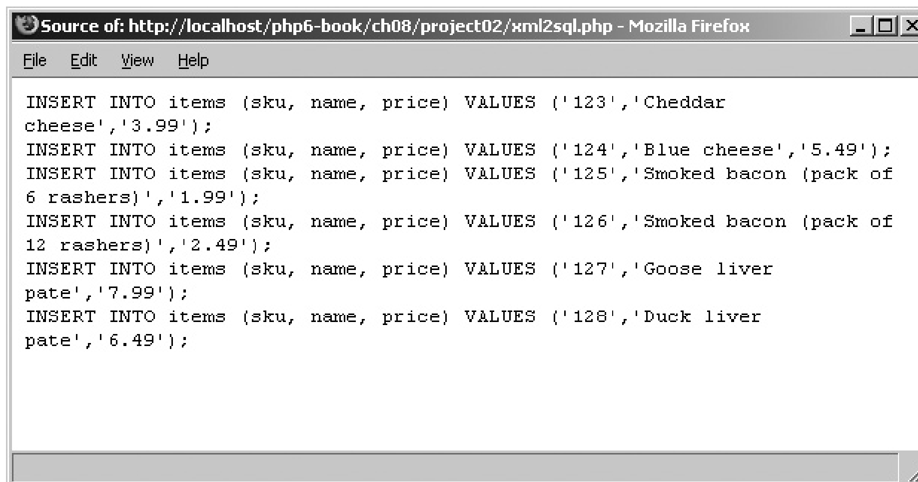

Figure 8-2 Converting XML to SQL with SimpleXML

Altering Element and Attribute Values

With SimpleXML, it's easy to change the content in an XML file: simply assign a new value to the corresponding object property using PHP's assignment operator (=). To illustrate, consider the following PHP script, which changes the title and author of the second book in *library.xml* and then outputs the revised XML document:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");

// change element values
// set new title and author for second book
$xml->book[1]->title = 'Invisible Prey';
$xml->book[1]->author = 'John Sandford';

// output new XML string
header('Content-Type: text/xml');
echo $xml->asXML();
?>
```

Here, SimpleXML is used to access the second `<book>` element by index, and the values of the `<title>` and `<author>` elements are altered by setting new values for the corresponding object properties. Notice the `asXML()` method, which is new in this example: it converts the nested hierarchy of SimpleXML objects and object properties back into a regular XML string.

Changing attribute values is just as easy: assign a new value to the corresponding key of the attribute array. Here's an example, which changes the sixth book's `'rating'` and outputs the result:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");

// change attribute values
// set new rating for sixth book
$xml->book[5]{'rating'} = 5;

// output new XML string
header('Content-Type: text/xml');
echo $xml->asXML();
?>
```

Adding New Elements and Attributes

In addition to allowing you to alter element and attribute values, SimpleXML also lets you dynamically add new elements and attributes to an existing XML document. To illustrate, consider the next script, which adds a new <book> to the *library.xml* XML data:

```
<?php
// load XML file
$xml = simplexml_load_file('library.xml') or die ("Unable to load XML!");

// get the last book 'id'
$numBooks = count($xml->book);
$lastID = $xml->book[($numBooks-1)]{'id'};

// add a new <book> element
$book = $xml->addChild('book');

// get the 'id' attribute
// for the new <book> element
// by incrementing $lastID by 1
$book->addAttribute('id', ($lastID+1));

// add 'rating' and 'genre' attributes
$book->addAttribute('genre', 'travel');
$book->addAttribute('rating', 5);

// add <title>, <author> and <page> elements
$title = $book->addChild('title', 'Frommer\'s Italy 2007');
$author = $book->addChild('author', 'Various');
$page = $book->addChild('pages', 820);

// output new XML string
header('Content-Type: text/xml');
echo $xml->asXML();
?>
```

Every SimpleXML object exposes an `addChild()` method (for adding new child elements) and an `addAttribute()` method (for adding new attributes). Both these methods accept a name and a value, generate the corresponding element or attribute, and attach it to the parent object within the XML hierarchy.

These methods are illustrated in the preceding listing, which begins by reading the existing XML document into a SimpleXML object. The root element of this XML document is stored in the PHP object `$xml`. The listing then needs to calculate the ID to be assigned to the new <book> element: it does this by counting the number of <book> elements already present in the XML document, accessing the last such element, retrieving that element's `'id'` attribute, and adding 1 to it.

With that formality out of the way, the listing then dives into element and attribute creation proper:

1. It starts off by attaching a new `<book>` element to the root element, by invoking the `$xml` object's `addChild()` method. This method accepts the name of the element to be created and (optionally) a value for that element. The resultant XML object is stored in the PHP object `$book`.
2. With the element created, it's now time to set its `'id'`, `'genre'`, and `'rating'` attributes. This is done via the `$book` object's `addAttribute()` method, which also accepts two arguments—the attribute name and value—and sets the corresponding associative array keys.
3. Once the outermost `<book>` element is fully defined, it's time to add the `<title>`, `<author>`, and `<pages>` elements as children of this `<book>` element. This is easily done by again invoking the `addChild()` method, this time of the `$book` object.
4. Once these child objects are defined, the object hierarchy is converted to an XML document string with the `asXML()` method.

Figure 8-3 illustrates what the output looks like.

Creating New XML Documents

You can also use SimpleXML to create a brand-spanking-new XML document from scratch, by initializing an empty SimpleXML object from an XML string, and then using the `addChild()` and `addAttribute()` methods to build the rest of the XML document tree. Consider the following example, which illustrates the process:

```
<?php
// load XML from string
$xmlStr = "<?xml version='1.0'?><person></person>";
$xml = simplexml_load_string($xmlStr);

// add attributes
$xml->addAttribute('age', '18');
$xml->addAttribute('sex', 'male');

// add child elements
$xml->addChild('name', 'John Doe');
$xml->addChild('dob', '04-04-1989');

// add second level of child elements
$address = $xml->addChild('address');
```

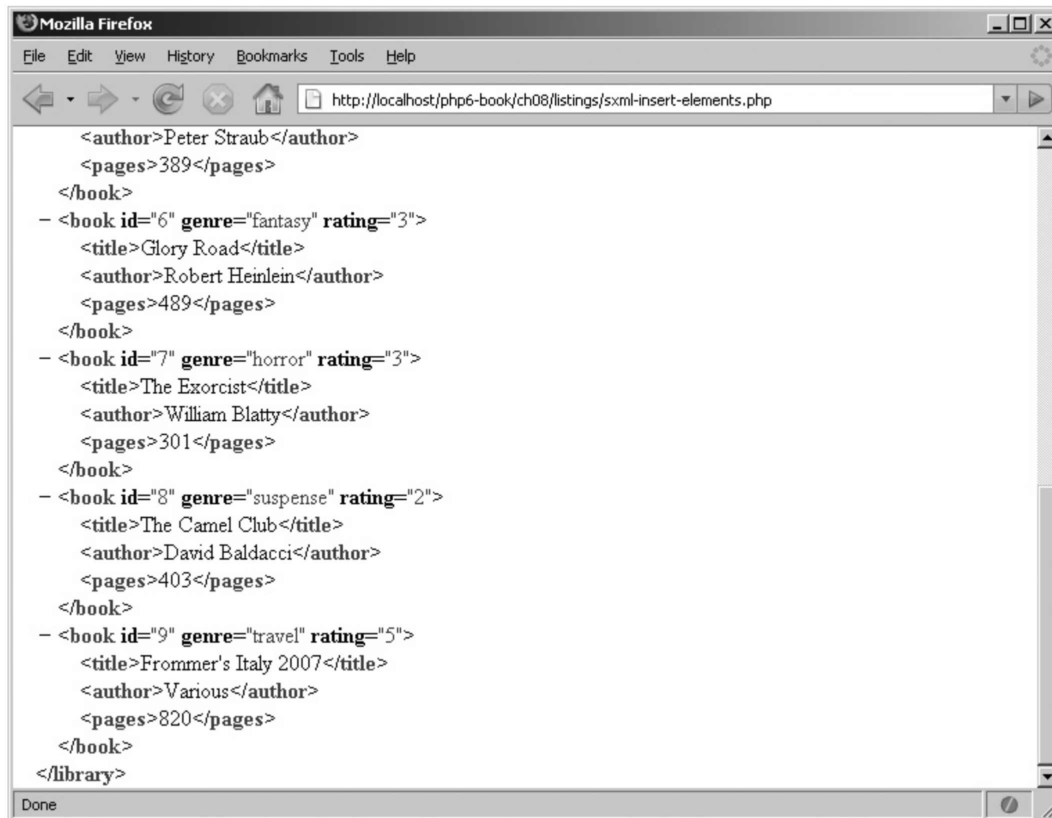

Figure 8-3 Inserting elements into an XML tree with SimpleXML

```
$address->addChild('street', '12 A Road');
$address->addChild('city', 'London');

// add third level of child elements
$country = $address->addChild('country', 'United Kingdom');
$country->addAttribute('code', 'UK');

// output new XML string
header('Content-Type: text/xml');
echo $xml->asXML();
?>
```

This PHP script is similar to what you've already seen in the preceding section, with one important difference: instead of grafting new elements and attributes on to a preexisting XML document tree, this one generates an XML document tree entirely from scratch!

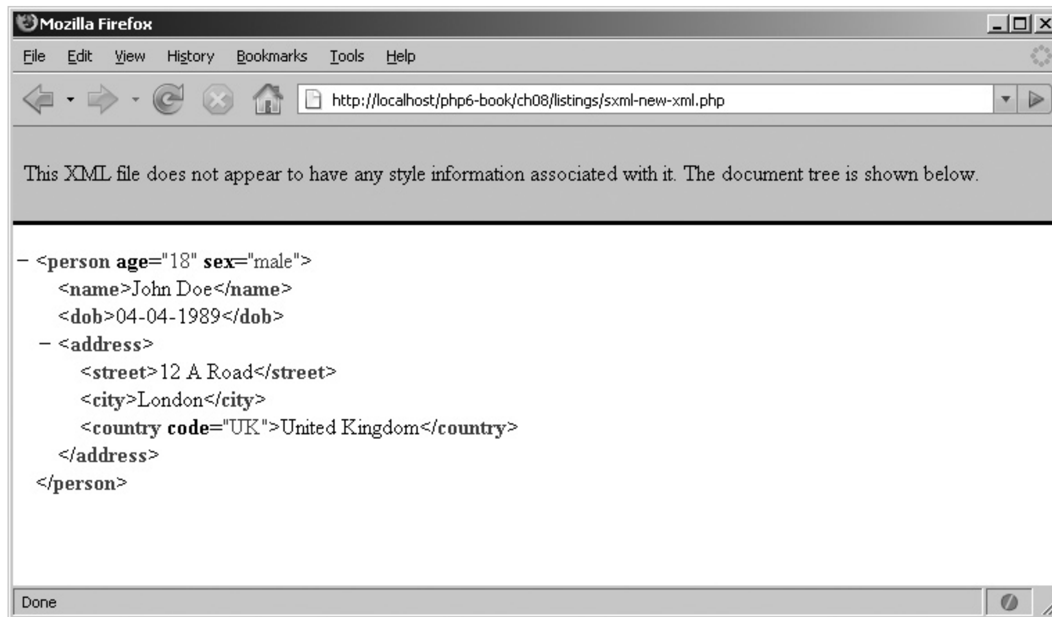

Figure 8-4 Dynamically generating a new XML document with SimpleXML

The script begins by initializing a string variable to hold the XML document prolog and root element. The `simplexml_load_string()` method takes care of converting this string into a SimpleXML object representing the document's root element. Once this object has been initialized, it's a simple matter to add child elements and attributes to it, and to build the rest of the XML document tree programmatically. Figure 8-4 shows the resulting XML document tree.

Try This 8-3 Reading RSS Feeds

RDF Site Summary (RSS) is an XML-based format originally devised by Netscape to distribute information about the content on its My.Netscape.com portal. Today, RSS is extremely popular on the Web as a way to distribute content; many Web sites offer RSS "feeds" that contain links and snippets of their latest news stories or content, and most browsers come with built-in RSS readers, which can be used to read and "subscribe" to these feeds.

An RSS document follows all the rules of XML markup and typically contains a list of resources (URLs), marked up with descriptive metadata. Here's an example:

```
<?xml version="1.0" encoding="utf-8"?>
<rss>
  <channel>
    <title>Feed title here</title>
    <link>Feed URL here</link>
    <description>Feed description here</description>
    <item>
      <title>Story title here</title>
      <description>Story description here</description>
      <link>Story URL here</link>
      <pubDate>Story timestamp here</pubDate>
    </item>
    <item>
      ...
    </item>
  </channel>
</rss>
```

As this sample document illustrates, an RSS document opens and closes with the `<rss>` element. A `<channel>` block contains general information about the Web site providing the feed; this is followed by multiple `<item>` elements, each of which represents a different content unit or news story. Each of these `<item>` elements further contains a title, a URL, and a description of the item.

Given this well-defined and hierarchical structure, parsing an RSS feed with SimpleXML is extremely simple. That's precisely what this next script does: it connects to a URL hosting a live RSS feed, retrieves the XML-encoded feed data, parses it, and converts it to an HTML page suitable for viewing in any Web browser. Here's the code (*rss2html.php*):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Project 8-3: Reading RSS Feeds</title>
    <style type="text/css">
    div.heading {
      font-weight: bolder;
    }
    div.story {
      background-color: white;
      border: 1px solid black;
```

(continued)

```
      width: 320px;
      height: 200px;
      margin: 20px;
    }
    div.headline a {
      font-weight: bolder;
      color: orange;
      margin: 5px;
    }
    div.body {
      margin: 5px;
    }
    div.timestamp {
      font-size: smaller;
      font-style: italic;
      margin: 5px;
    }
    ul {
       list-style-type: none;
    }
    li {
       float: left;
    }
    </style>
  </head>
  <body>
    <h2>Project 8-3: Reading RSS Feeds</h2>
<?php
// read newsvine.com's RSS feed for top technology news stories
$xml = simplexml_load_file("http://www.newsvine.com/_feeds/rss2
 /tag?id=technology") or die("ERROR: Cannot read RSS feed");
?>
    <h3 style="heading"><?php echo $xml->channel->title; ?></h3>
    <ul>
<?php
// iterate over list of stories
// print each story's title, URL and timestamp
// and then the story body
foreach ($xml->channel->item as $item) {
?>
      <li>
        <div class="story">
          <div class="headline">
            <a href="<?php echo $item->link; ?>">
              <?php echo $item->title; ?>
            </a>
          </div>
```

```
          <div class="timestamp"><?php echo $item->pubDate; ?></div>
          <div class="body"><?php echo $item->description; ?></div>
        </div>
      </li>
<?php
}
?>
    </ul>
  </body>
</html>
```

This script begins by using SimpleXML's `simplexml_load_file()` method to connect to a remote URL—in this case, an RSS feed hosted by NewsVine.com—and convert the XML data found therein to a SimpleXML object. It then uses SimpleXML's ability to loop over node collections to quickly retrieve each news story's title, URL, timestamp, and body; marks these bits of information up with HTML; and prints them to the page.

Figure 8-5 illustrates what the output might look like.

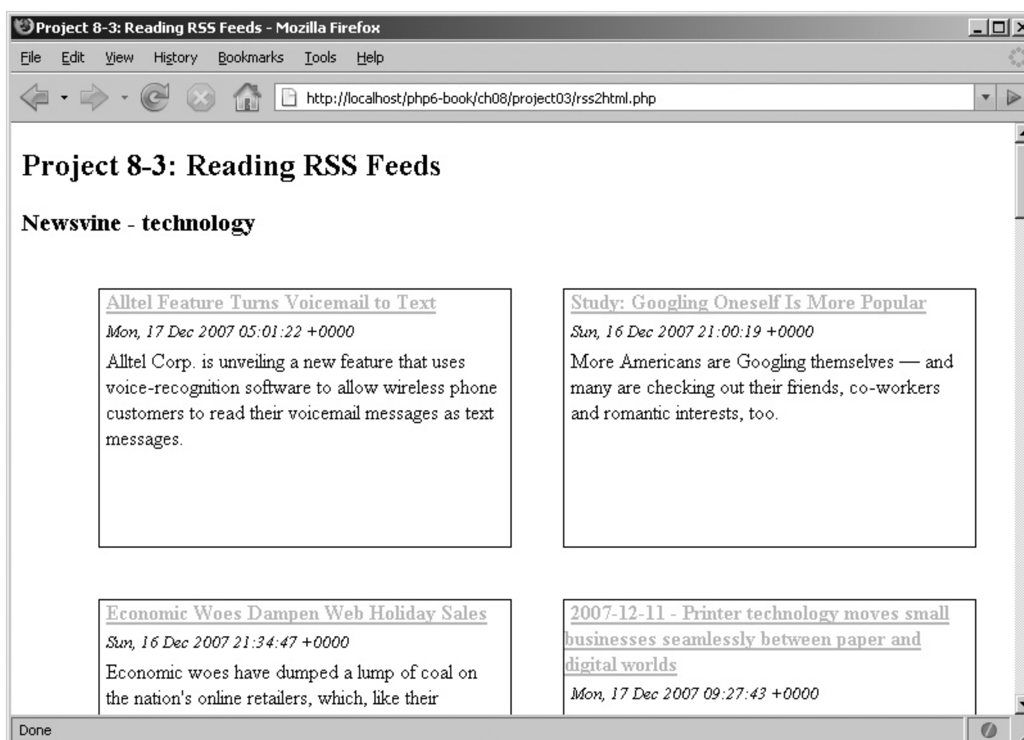

Figure 8-5 Parsing an RSS feed with SimpleXML

Using PHP's DOM Extension

Now, while PHP's SimpleXML extension is easy to use and understand, it's not very good for anything other than the most basic XML manipulation. For more complex XML operations, it's necessary to look further afield, to PHP's DOM extension. This extension, which is also enabled by default in PHP 5, provides a sophisticated toolkit that complies with the DOM Level 3 standard and brings comprehensive XML parsing capabilities to PHP.

Working with Elements

The DOM parser works by reading an XML document and creating objects to represent the different parts of that document. Each of these objects comes with specific methods and properties, which can be used to manipulate and access information about it. Thus, the entire XML document is represented as a "tree" of these objects, with the DOM parser providing a simple API to move between the different branches of the tree.

To illustrate how this works in practice, let's revisit the *address.xml* file from the preceding section:

```
<?xml version='1.0'?>
<address>
  <street>13 High Street</street>
  <county>Oxfordshire</county>
  <city>
    <name>Oxford</name>
    <zip>OX1 1BA</zip>
  </city>
  <country>UK</country>
</address>
```

Here's a PHP script that uses the DOM extension to parse this file and retrieve the various components of the address:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('address.xml');

// get root element
$root = $doc->firstChild;
```

```
// get text node 'UK'
echo "Country: " . $root->childNodes->item(3)->nodeValue . "\n";

// get text node 'Oxford'
echo "City: " . $root->childNodes->item(2)->childNodes->
 item(0)->nodeValue . "\n";

// get text node 'OX1 1BA'
echo "Postal code: " . $root->childNodes->item(2)->childNodes->
 item(1)->nodeValue . "\n";

// output: 'Country: UK \n City: Oxford \n Postal code: OX1 1BA'
?>
```

A quick glance, and it's clear that we're not in SimpleXML territory any longer. With PHP's DOM extension, the first step is always to initialize an instance of the DOMDocument object, which represents an XML document. Once this object has been initialized, it can be used to parse an XML file via its `load()` method, which accepts the disk path to the target XML file.

The result of the `load()` method is a tree containing DOMNode objects, with every object exposing various properties and methods for accessing its parent, child, and sibling nodes. For example, every DOMNode object exposes a `parentNode` property, which can be used to access its parent node, and a `childNodes` property, which returns a collection of its child nodes. In a similar vein, every DOMNode object also exposes `nodeName` and `nodeValue` properties, which can be used to access the node's name and value respectively. It's thus quite easy to navigate from node to node of the tree, retrieving node values at each stage.

To illustrate the process, consider the preceding script carefully. Once the XML document has been `load()`-ed, it calls the DOMDocument object's `firstChild` property, which returns a DOMNode object representing the root element `<address>`. This DOMNode object, in turn, has a `childNodes` property, which returns a collection of all the child elements of `<address>`. Individual elements of this collection can be accessed via their index position using the `item()` method, with indexing starting from zero. These elements are again represented as DOMNode objects; as such, their names and values are therefore accessible via their `nodeName` and `nodeValue` properties.

Thus, the element `<country>`, which is the fourth child element under `<address>`, is accessible via the path `$root->childNodes->item(3)`, and the text value of this element, `'UK'`, is accessible via the path `$root->childNodes->item(3)->nodeValue`. Similarly, the element `<name>`, which is the first child of the `<city>` element, is accessible via the path `$root->childNodes->item(2)->childNodes->item(0)`, and the text value `'Oxford'` is accessible via the path `$root->childNodes->item(2)->childNodes->item(0)->nodeValue`.

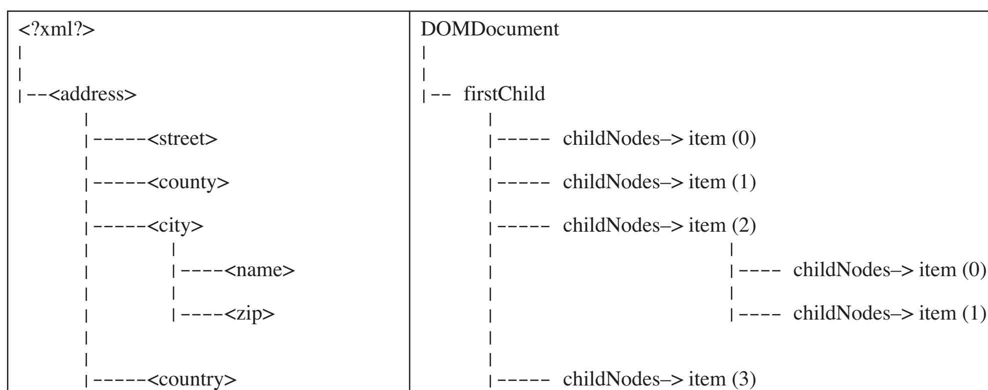

Figure 8-6 DOM relationships

Figure 8-6 should make these relationships clearer, by mapping the XML document tree from *address.xml* to the DOM methods and properties used in this section.

Ask the Expert

Q: When I process an XML document using the DOM, there often appear to be extra text nodes in each node collection. However, when I access these nodes, they appear to be empty. What's going on?

A: As per the DOM specification, all document whitespace, including carriage returns, must be treated as a text node. If your XML document contains extra whitespace, or if your XML elements are neatly formatted and indented on separate lines, this whitespace will be represented in your node collections as apparently empty text nodes. In the PHP DOM API, you can disable this behavior by setting the `DOMDocument->preserveWhiteSpace` property to `'false'`, as the examples in this section do.

An alternative approach—and one that can come in handy when you're faced with a deeply nested XML tree—is to use the DOMDocument object's `getElementsByTagName()` method to directly retrieve all elements with a particular name. The output of this method is a collection of matching DOMNode objects; it's then easy to iterate over the collection with a `foreach` loop and retrieve the value of each node.

If your document happens to have only one instance of each element—as is the case with *address.xml*—using `getElementsByTagName()` can serve as an effective shortcut to the traditional tree navigation approach. Consider the following example, which produces the same output as the preceding listing using this approach:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('address.xml');

// get collection of <country> elements
$country = $doc->getElementsByTagName('country');
echo "Country: " . $country->item(0)->nodeValue . "\n";

// get collection of <name> elements
$city = $doc->getElementsByTagName('name');
echo "City: " . $city->item(0)->nodeValue . "\n";

// get collection of <zip> elements
$zip = $doc->getElementsByTagName('zip');
echo "Postal code: " . $zip->item(0)->nodeValue . "\n";

// output: 'Country: UK \n City: Oxford \n Postal code: OX1 1BA'
?>
```

In this example, the `getElementsByTagName()` method is used to return a DOMNode collection representing all elements with the name `<country>` in the first instance. From the XML document tree, it's clear that the collection will contain only one DOMNode object. Accessing the value of this node is then simply a matter of calling the collection's `item()` method with argument 0 (for the first index position) to get the DOMNode object, and then reading its `nodeValue` property.

In most cases, however, your XML document will not have only one instance of each element. Take, for example, the *library.xml* file you've seen in previous sections, which contains multiple instances of the `<book>` element. Even in such situations, the `getElementsByTagName()` method is useful to quickly and efficiently create a subset of matching nodes, which can be processed using a PHP loop. To illustrate, consider

this next example, which reads *library.xml* and prints the title and author names found within it:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');

// get collection of <book> elements
// for each <book>, get the value of the <title> and <author> elements
// output: 'The Shining is written by Stephen King. \n ...'
$books = $doc->getElementsByTagName('book');
foreach ($books as $book) {
  $title = $book->getElementsByTagName('title')->item(0)->nodeValue;
  $author = $book->getElementsByTagName('author')->item(0)->nodeValue;
  echo  "$title is written by $author.\n";
}
?>
```

In this case, the first call to `getElementsByTagName()` returns a collection representing all the `<book>` elements from the XML document. It's then easy to iterate over this collection with a `foreach()` loop, processing each DOMNode object and retrieving the value of the corresponding `<title>` and `<author>` elements with further calls to `getElementsByTagName()`.

TIP

You can return a collection of all the elements in a document by calling `DOMDocument-> getElementsByTagName(*)`.

To find out how many elements were returned by a call to `getElementsByTagName()`, use the resulting collection's `length` property. Here's an example:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;
```

```
// read XML file
$doc->load('library.xml');

// get collection of <book> elements
// return a count of the total number of <book> elements
// output: '8 book(s) found.'
$books = $doc->getElementsByTagName('book');
echo $books->length . ' book(s) found.';
?>
```

Working with Attributes

The DOM also includes extensive support for attributes: every DOMElement object comes with a `getAttribute()` method, which accepts an attribute name and returns the corresponding value. Here's an example, which prints each book's rating and genre from *library.xml*:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');

// get collection of <book> elements
// for each book
// retrieve and print 'genre' and 'rating' attributes
// output: 'The Shining \n Genre: horror \n Rating: 5 \n\n ...'
$books = $doc->getElementsByTagName('book');
foreach ($books as $book) {
  $title = $book->getElementsByTagName('title')->item(0)->nodeValue;
  $rating = $book->getAttribute('rating');
  $genre = $book->getAttribute('genre');
  echo "$title\n";
  echo "Genre: $genre\n";
  echo "Rating: $rating\n\n";
}
?>
```

What if you don't know the attribute name but simply want to process all attributes of an element? Well, every DOMElement has an `attributes` property, which returns a collection of all the element's attributes. It's easy to iterate over this collection to retrieve

each attribute's `name` and `value`. The following example demonstrates, by revising the preceding script to use this approach:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');

// get collection of <book> elements
// for each book
// retrieve and print all attributes
// output: 'The Shining \n id: 1 \n genre: horror \n rating: 5 \n\n ...'
$books = $doc->getElementsByTagName('book');
foreach ($books as $book) {
  $title = $book->getElementsByTagName('title')->item(0)->nodeValue;
  echo "$title\n";
  foreach ($book->attributes as $attr) {
    echo "$attr->name: $attr->value \n";
  }
  echo "\n";
}
?>
```

Try This 8-4 Recursively Processing an XML Document Tree

If you plan to work with XML and PHP in the future, this next project will almost certainly come in handy some day: it's a simple program that starts at the root of the XML document tree and works its way through to the ends of its branches, processing every element and attribute it finds on the way. Given the tree-like nature of an XML document, the most efficient way to accomplish this task is with a recursive function—and given the wealth of information supplied by the DOM, writing such a function is quite easy.

Assume for a moment that the XML document to be processed looks like this (*inventory.xml*):

```
<?xml version='1.0'?>
<objects>
  <object color="red" shape="square">
```

```
    <length units="cm">5</length>
  </object>
  <object color="red" shape="circle">
    <radius units="px">7</radius>
  </object>
  <object color="green" shape="triangle">
    <base units="in">1</base>
    <height units="in">2</height>
  </object>
  <object color="blue" shape="triangle">
    <base units="mm">100</base>
    <height units="mm">50</height>
  </object>
  <object color="yellow" shape="circle">
    <radius units="cm">18</radius>
  </object>
</objects>
```

And here's the PHP code to recursively process this (or any other) XML document using the DOM:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Project 8-4: Recursively Processing An XML Document</title>
  </head>
  <body>
    <h2>Project 8-4: Recursively Processing An XML Document</h2>
    <pre>
<?php
// recursive function to process XML node collection
function xmlProcess($node, $depthMarker) {

  // process this node's children
  foreach ($node->childNodes as $n) {
    switch ($n->nodeType) {

      // for elements, print element name
      case XML_ELEMENT_NODE:
        echo "$depthMarker <b>$n->nodeName</b> \n";
        // if the element has attributes
        // list their names and values
        if ($n->attributes->length > 0) {
          foreach ($n->attributes as $attr) {
            echo "$depthMarker  <i>attr</i>: $attr->name => $attr->value \n";
          }
        }
        break;
```

(continued)

```
      // for text data, print value
      case XML_TEXT_NODE:
        echo "$depthMarker <i>text</i>: \"$n->nodeValue\" \n";
        break;
    }

    // if this node has a further level of sub-nodes
    // increment depth marker
    // run recursively
    if ($n->hasChildNodes()) {
      xmlProcess($n, $depthMarker . DEPTH_CHAR);
    }
  }
}
// end function definition

// define the character used for indentation
define ('DEPTH_CHAR', ' ');

// initialize DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('objects.xml');

// call recursive function with root element
xmlProcess($doc->firstChild, DEPTH_CHAR);
?>
    </pre>
  </body>
</html>
```

In this program, the user-defined `xmlProcess()` function is a recursive function that works by accepting a DOMNode object as input, retrieving a collection of this node's children by reading the object's `childNodes` property, and iterating over this collection with a `foreach` loop. Depending on whether the current node is an element node or a text node, it prints either the node name or the node value. If the node is an element node, it performs an additional step of checking for attributes and printing those as necessary. A "depth string" is used to indicate the hierarchical position of the node in the output; this string is automatically incremented every time the loop runs.

Having completed all these tasks, the last action of the function is to check whether the current node has any children; if it does, it calls itself recursively to process the next level of the node tree. The process continues until no further nodes remain to be processed.

Figure 8-7 illustrates the output of the program when `xmlProcess()` is called with the document's root element as input argument.

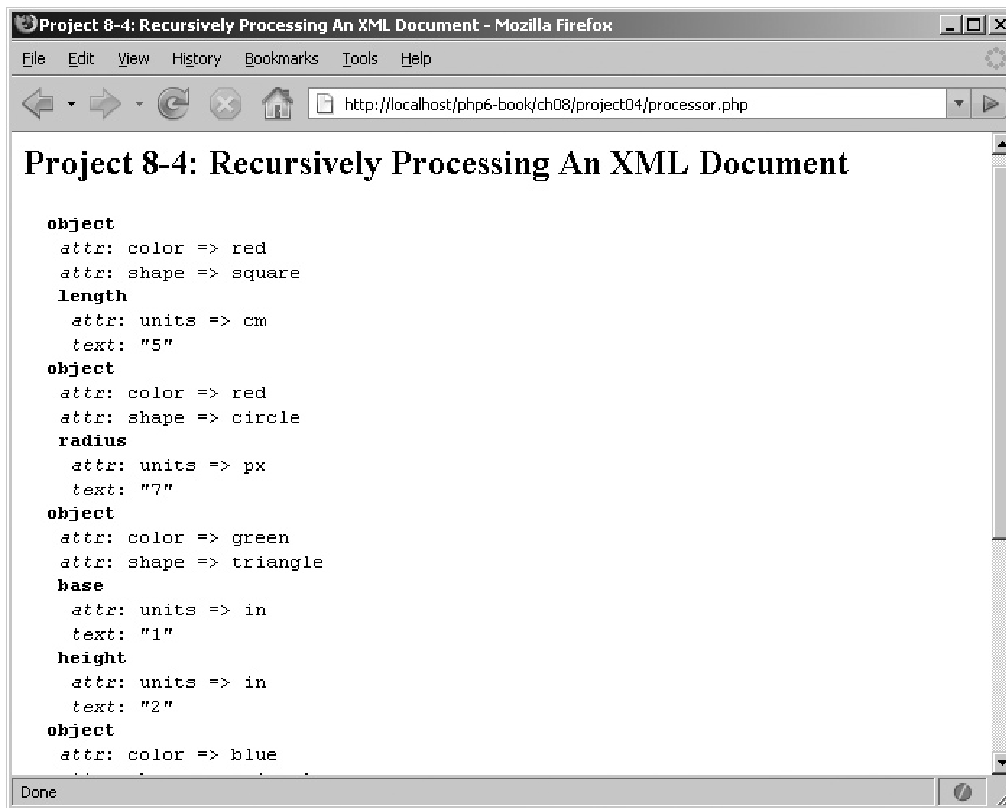

Figure 8-7 Recursively processing an XML document with the DOM

Altering Element and Attribute Values

Under the DOM, changing the value of an XML element is quite simple: navigate to the DOMNode object representing the element and alter its `nodeValue` property to reflect the new value. To illustrate, consider the following PHP script, which changes the title and author of the second book in *library.xml,* and then outputs the revised XML document:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');
```

```
// get collection of <book> elements
$books = $doc->getElementsByTagName('book');

// change the <title> element of the second <book>
$books->item(1)->getElementsByTagName('title')->item(0)->nodeValue =
'Invisible Prey';

// change the <author> element of the second <book>
$books->item(1)->getElementsByTagName('author')->item(0)->nodeValue =
'John Sandford';

// output new XML string
header('Content-Type: text/xml');
echo $doc->saveXML();
?>
```

Here, the `getElementsByTagName()` method is used to first obtain a collection of `<book>` elements and navigate to the second element in this collection (index position: 1). It's then used again, to obtain references to DOMNode objects representing the `<title>` and `<author>` element. The `nodeValue` properties of these objects are then assigned new values using PHP's assignment operator, and the revised XML tree is converted back into a string with the DOMDocument object's `saveXML()` method.

Changing attribute values is just as easy: assign a new value to an attribute using the corresponding DOMElement object's `setAttribute()` method. Here's an example, which changes the sixth book's `'rating'` and outputs the result:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');

// get collection of <book> elements
$books = $doc->getElementsByTagName('book');

// change the 'genre' element of the fifth <book>
$books->item(4)->setAttribute('genre', 'horror-suspense');

// output new XML string
header('Content-Type: text/xml');
echo $doc->saveXML();
?>
```

Creating New XML Documents

The DOM comes with a full-fledged API for creating new XML documents, or for grafting elements, attributes, and other XML structures on to an existing XML document tree. This API, which is much more sophisticated than that offered by SimpleXML, should be your first choice when dynamically creating or modifying an XML document tree through PHP.

The best way to illustrate this API is with an example. Consider the following script, which sets up a new XML file from scratch:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument('1.0');

// create and attach root element <schedule>
$root = $doc->createElement('schedule');
$schedule = $doc->appendChild($root);

// create and attach <course> element under <schedule>
$course = $doc->createElement('course');
$schedule->appendChild($course);

// create and attach <title> element under <course>
// add a value for the <title> element
$title = $doc->createElement('title');
$titleData = $doc->createTextNode('Macro-Economics');
$course->appendChild($title);
$title->appendChild($titleData);

// create and attach <teacher> element under <course>
// add a value for the <teacher> element
$teacher = $doc->createElement('teacher');
$teacherData = $doc->createTextNode('Professor Q. Draw');
$course->appendChild($teacher);
$teacher->appendChild($teacherData);

// create and attach <credits> element under <course>
// add a value for the <credits> element
$credits = $doc->createElement('credits');
$creditData = $doc->createTextNode('4');
$course->appendChild($credits);
$credits->appendChild($creditData);

// attach an attribute 'transferable' to the <credits> element
// set a value for the attribute
$transferable = $doc->createAttribute('transferable');
```

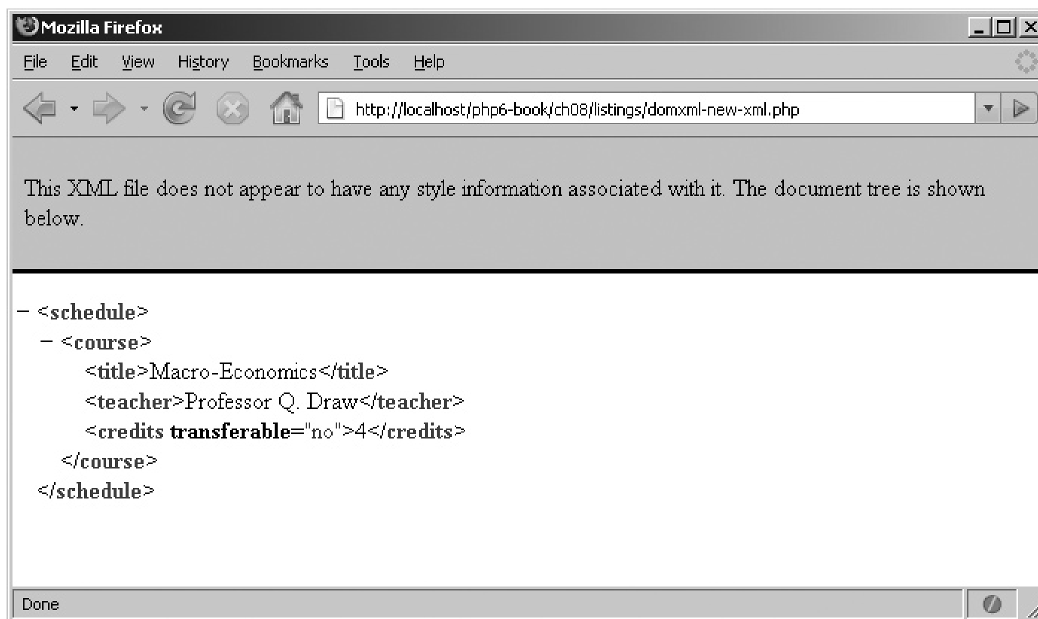

Figure 8-8 Dynamically generating a new XML document with the DOM

```
$credits->appendChild($transferable);
$credits->setAttribute('transferable', 'no');

// format XML output
$doc->formatOutput = true;

// output new XML string
header('Content-Type: text/xml');
echo $doc->saveXML();
?>
```

Figure 8-8 illustrates the XML document generated by this script.

This script introduces some new methods, all related to dynamically creating XML nodes and attaching them to an XML document tree. There are two basic steps involved in this process:

1. Create an object representing the XML structure you wish to add. The base DOMDocument object exposes `create...()` methods corresponding to each of the primary XML structures: `createElement()` for element objects, `createAttribute()` for attribute objects, and `createTextNode()` for character data.

2. Attach the newly minted object at the appropriate point in the document tree, by calling the parent's `appendChild()` method.

The previous listing illustrates these steps, following a specific sequence to arrive at the result tree shown in Figure 8-8.

1. It begins by first initializing a DOMDocument object named `$doc` and then calling its `createElement()` method to generate a new DOMElement object named `$schedule`. This object represents the document's root element; as such, it is attached to the base of the DOM tree by calling the `$doc->appendChild()` method.
2. One level below the root `<schedule>` element comes a `<course>` element. In DOM terms, this is accomplished by creating a new DOMElement object named `$course` with the DOMDocument object's `createElement()` method, and then attaching this object to the tree under the `<schedule>` element by calling `$schedule->appendChild()`.
3. One level below the `<course>` element comes a `<title>` element. Again, this is accomplished by creating a DOMElement object named `$title` and then attaching this object under `<course>` by calling `$course->appendChild()`. There's a twist here, though: the `<title>` element contains the text value `'Macro-Economics'`. To create this text value, the script creates a new DOMTextNode object via the `createTextNode()` object, populates it with the text string, and then attaches it as a child of the `<title>` element by calling `$title->appendChild()`.
4. The same thing happens a little further along, when creating the `<credits>` element. Once the element and its text value have been defined and attached to the document tree under the `<course>` element, the `createAttribute()` method is used to create a new DOMAttr object to represent the attribute `'transferable'`. This attribute is then attached to the `<credits>` element by calling `$credits->appendChild()`, and a value is assigned to the attribute in the normal fashion, by calling `$credits->setAttribute()`.

Converting Between DOM and SimpleXML

An interesting feature in PHP is the ability to convert XML data between DOM and SimpleXML. This is accomplished by means of two functions: the `simplexml_import_dom()` function, which accepts a DOMElement object and returns a SimpleXML object,

and the `dom_import_simplexml()` function, which does the reverse. The following example illustrates this interoperability:

```
<?php
// initialize new DOMDocument
$doc = new DOMDocument();

// disable whitespace-only text nodes
$doc->preserveWhiteSpace = false;

// read XML file
$doc->load('library.xml');

// get collection of <book> elements
$books = $doc->getElementsByTagName('book');

// convert the sixth <book> to a SimpleXML object
// print title of sixth book
// output: 'Glory Road'
$sxml = simplexml_import_dom($books->item(5));
echo $sxml->title;
?>
```

Try This 8-5 Reading and Writing XML Configuration Files

Now that you know how to read and create XML document trees programmatically, let's use this knowledge in an application that's increasingly popular these days: XML-based configuration files, which use XML to mark up an application's configuration data.

The next listing illustrates this in action, generating a Web form that allows users to configure an oven online by entering configuration value for temperature, mode, and heat source. When the form is submitted, the data entered by the user is converted to XML and saved to a disk file. When users revisit the form, the data previously saved to the file is read and used to prefill the form's fields.

Here's the code (*configure.php*):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Project 8-5: Reading And Writing XML Configuration Files</title>
  </head>
```

```
  <body>
    <h2>Project 8-5: Reading And Writing XML Configuration Files</h2>
    <h3 style="background-color: silver">Oven Configuration</h3>
<?php
    // define configuration file name and path
    $configFile = 'config.xml';

    // if form not yet submitted
    // display form
    if (!isset($_POST['submit'])) {

      // set up array with default parameters
      $data = array();
      $data['mode'] = null;
      $data['temperature'] = null;
      $data['duration'] = null;
      $data['direction'] = null;
      $data['autooff'] = null;

      // read current configuration values
      // use them to pre-fill the form
      if (file_exists($configFile)) {
        $doc = new DOMDocument();
        $doc->preserveWhiteSpace = false;
        $doc->load($configFile);
        $oven = $doc->getElementsByTagName('oven');
        foreach ($oven->item(0)->childNodes as $node) {
          $data[$node->nodeName] = $node->nodeValue;
        }
      }
?>
    <form method="post" action="configure.php">
      Mode: <br />
      <select name="data[mode]">
        <option value="grill" <?php echo ($data['mode'] == 'grill') ?
'selected' : null; ?>>Grill</option>
        <option value="bake" <?php echo ($data['mode'] == 'bake') ?
'selected' : null; ?>>Bake</option>
        <option value="toast" <?php echo ($data['mode'] == 'toast') ?
'selected' : null; ?>>Toast</option>
      </select>

      <p>

      Temperature: <br />
      <input type="text" size="2" name="data[temperature]"  value="<?php echo
$data['temperature']; ?>"/>

      <p>
```

(continued)

```
      Duration (minutes): <br />
      <input type="text" size="2" name="data[duration]" value="<?php echo
$data['duration']; ?>"/>

      <p>

      Heat source and direction: <br />
      <input type="radio" name="data[direction]" value="top-down" <?php echo
($data['direction'] == 'top-down') ? 'checked' : null; ?>>Top, downwards</input>
      <input type="radio" name="data[direction]" value="bottom-up" <?php echo
($data['direction'] == 'bottom-up') ? 'checked' : null; ?>>Bottom, upwards
</input>
      <input type="radio" name="data[direction]" value="both" <?php echo
($data['direction'] == 'both') ? 'checked' : null; ?>>Both</input>

      <p>

      Automatically power off when done:
      <input type="checkbox" name="data[autooff]" value="yes" <?php echo
($data['autooff'] == 'yes') ? 'checked' : null; ?>/>

      <p>

      <input type="submit" name="submit" value="Submit" />
    </form>
<?php
    // if form submitted
    // process form input
    } else {
      // read submitted data
      $config = $_POST['data'];

      // validate submitted data as necessary

      if ((trim($config['temperature']) == '') || (trim($config['temperature'])
!= '' && (int)$config['temperature'] <= 0)) {
        die('ERROR: Please enter a valid oven temperature');
      }

      if ((trim($config['duration']) == '') || (trim($config['duration']) != ''
&& (int)$config['duration'] <= 0)) {
        die('ERROR: Please enter a valid duration');
      }

      // generate new XML document
      $doc = new DOMDocument();

      // create and attach root element <configuration>
      $root = $doc->createElement('configuration');
      $configuration = $doc->appendChild($root);
```

```
      // create and attach <oven> element under <schedule>
      $oven = $doc->createElement('oven');
      $configuration->appendChild($oven);

      // write each configuration value to the file
      foreach ($config as $key => $value) {
        if (trim($value) != '') {
          $elem = $doc->createElement($key);
          $text = $doc->createTextNode($value);
          $oven->appendChild($elem);
          $elem->appendChild($text);
        }
      }

      // format XML output
      // save XML file
      $doc->formatOutput = true;
      $doc->save($configFile) or die('ERROR: Cannot write configuration file');
      echo 'Configuration data successfully written to file.';
    }
?>
  </body>
</html>
```

Figure 8-9 illustrates the Web form generated by this script.

Once this form is submitted, the data entered into it arrives in the form of an associative array, whose keys correspond to XML element names. This data is first validated, and the DOM API is then used to generate a new XML document tree containing these elements and their values. Once the tree is completely generated, the DOMDocument object's `save()` function is used to write the XML to a disk file.

Here's an example of what the XML output file *config.xml* would look like after submitting the form in Figure 8-9:

```
<?xml version="1.0"?>
<configuration>
  <oven>
    <mode>toast</mode>
    <temperature>22</temperature>
    <duration>1</duration>
    <direction>bottom-up</direction>
    <autooff>yes</autooff>
  </oven>
</configuration>
```

If a user revisits the Web form, the script first checks if a configuration file named *config.xml* exists in the current directory. If it does, the XML data in the file is read into a new DOMDocument object with the `load()` method and converted into an associative

(continued)

Figure 8-9 A Web form for configuration data

array by iterating over the list of child nodes in a loop. The various radio buttons, check boxes, and selection lists in the form are then checked or preselected, depending on the values in this array.

Figure 8-10 illustrates the form, prefilled with data read from the XML configuration file.

If the user submits the Web form with new values, these new values will again be encoded in XML and used to rewrite the configuration file. Because the configuration is expressed in XML, any application that has XML parsing capabilities can read and use this data. XML, when used in this fashion, thus provides a way to transfer information between applications, even if they're written in different programming languages or run on incompatible operating systems.

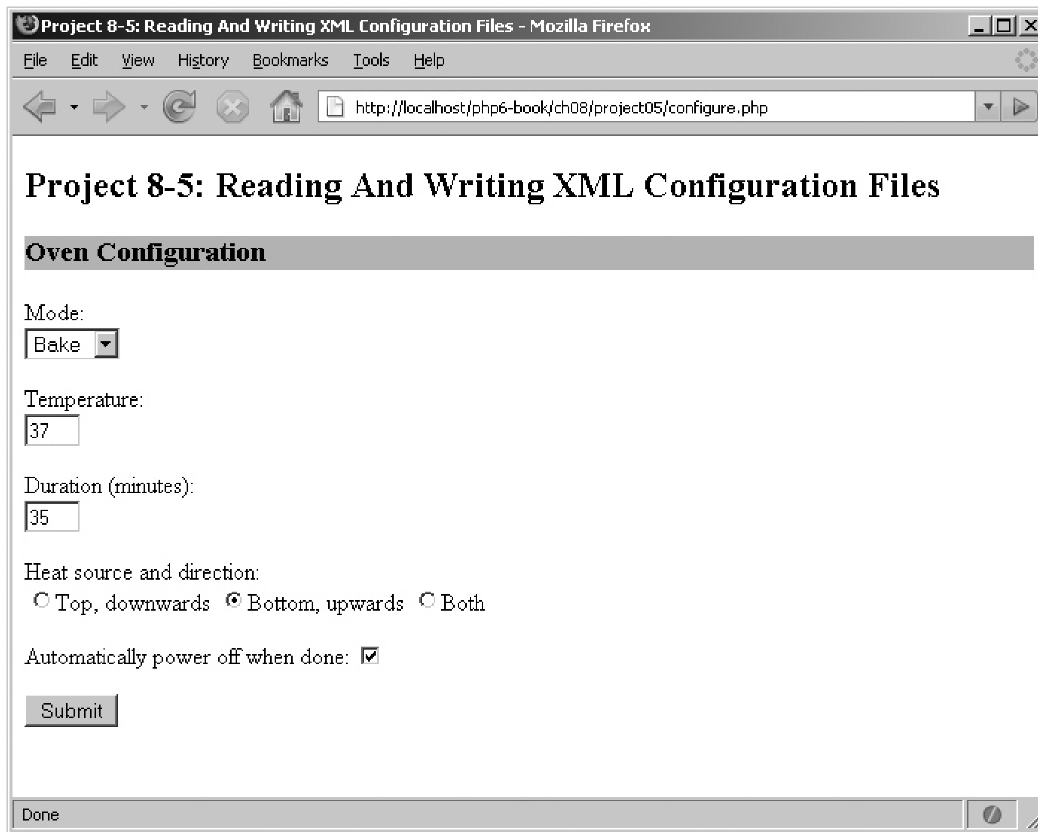

Figure 8-10 The same Web form, prefilled with configuration data

Summary

At the end of this chapter, you should know enough to begin writing PHP programs that can successfully interact with XML-encoded data. This chapter began with an introduction to XML, explaining basic XML structures like elements, attributes, and character data, and providing a crash course in XML technologies and parsing methods. It then proceeded into a discussion of two of PHP's most popular extensions for XML processing, the SimpleXML and DOM extensions, and showed you how each of these extensions could be used to access element and attribute values, create node collections, and programmatically generate or change XML document trees. Various projects, ranging from an XML-to-SQL converter to an RSS feed parser, were used to illustrate practical applications of the interface between XML and PHP.

XML is an extensive topic, and the material in this chapter barely begins to scratch its surface. However, there are many excellent tutorials and articles on XML and PHP on the Web, and links to some of these are presented here, should you be interested in learning more about this interesting and continually changing field:

- XML basics, at **www.melonfire.com/community/columns/trog/article.php?id=78** and **www.melonfire.com/community/columns/trog/article.php?id=79**
- XPath basics, at **www.melonfire.com/community/columns/trog/article.php?id=83**
- XSL basics, at **www.melonfire.com/community/columns/trog/article.php?id=82** and **www.melonfire.com/community/columns/trog/article.php?id=85**
- SimpleXML functions, at **www.php.net/simplexml**
- DOM API functions in PHP, at **www.php.net/dom**
- The DOM specification, at **www.w3.org/DOM/**
- Building XML documents using PHP and PEAR, at **www.melonfire.com/community/columns/trog/article.php?id=180**
- Serializing XML, at **www.melonfire.com/community/columns/trog/article.php?id=244**
- Performing XML-based Remote Procedure Calls (RPC) with PHP, at **www.melonfire.com/community/columns/trog/article.php?id=274**

Chapter 8 Self Test

1. What are the two methods of parsing an XML document, and how do they differ?
2. Name two characteristics of a well-formed XML document.
3. Given the following XML document (*email.xml*), write a program to retrieve and print all the e-mail addresses from the document using SimpleXML:

```
<?xml version='1.0'?>
<data>
  <person>
    <name>Clone One</name>
    <email>one@domain.com</name>
  </person>
  <person>
    <name>Clone SixtyFour</name>
    <email>sixtyfour@domain.com</name>
  </person>
```

```
  <person>
    <name>Clone Three</name>
    <email>three@domain.com</name>
  </person>
  <person>
    <name>Clone NinetyNine</name>
    <email>ninetynine@domain.com</name>
  </person>
</data>
```

4. Given the following XML document (*tree.xml*), suggest three different ways to retrieve the text value `'John'` using the DOM:

```
<?xml version='1.0'?>
<tree>
  <person type="grandpa" />
  <person type="grandma" />
  <children>
    <person type="pa" />
    <person type="ma" />
    <children>
      <person type="bro">
        <name>John</name>
      </person>
      <person type="sis">
        <name>Jane</name>
      </person>
    </children>
  </children>
</tree>
```

5. Write a program to count the number of elements in an XML file. Use the DOM.

6. Write a program to process the *library.xml* file from earlier in this chapter, increase each book's rating by 1, and print the revised output. Use SimpleXML.

7. Write a program that connects to a MySQL database and retrieves the contents of any one of its tables as an XML file. Use the DOM.