XML and PHP 5

Douglas Putnam

November 1, 2006

Contents

Abstract

XML is living up to its touted role as a universal language for communication between platforms. It has been called the ”new web revolution.” XML has been used as a database for storing documents, but data storage is not its primary purpose. It was developed to pass information from one system to another in a common format. PHP 5’s SimpleXML extension is currently the simplest way to work with XML.

1 Introduction

XML is a tagged language, with data contained in structured, tagged elements of the document. The XML document must be parsed to extract the information. This information is often converted into another format. In this lecture we will focus on using PHP to read and transform XML documents, and on using XML as a communication protocol with Remote Services. XML is a big topic and we will not cover everything.

In this lecture we will learn:

- The structure of an XML document
- The terminology needed to work with XML documents
- How to parse an XML document using the two mainstream methods: SAX and DOM
- How to parse a simple XML document using the PHP SimpleXML extension
- How to use some handy PEAR packages for XML
- How to convert an XML document into another format using XSLT
- How to share information between systems using XML

2 XML Terminology

As with any area of computer science, XML has its own terminology, and some of it is likely be unfamiliar. Below is a typical XML document:

Example 1

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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" xml:lang="en" lang="en">
  <head>
    <title>XML Example</title>
  </head>
  <body background="bg.png">
    <p>
      Try  PHP at CCSF <a href="http://norbert.ccsf.edu/cs130b">here</a>.
      <br />
      Thanks for all the fish!
    </p>
  </body>
</html>
```

XML Declaration The first line of the sample script is the XML declaration, which specifies the XML version and the XML file encoding. This declaration starts with `<?`. As we know, PHP regards these characters as the beginning of a section of PHP code. This can cause problems if we an XML document as a PHP script if PHP short tags are enabled (the default). PHP will read the `<?` as the opening tag of a PHP section. If you plan to work with XML together with PHP, you can change the `short_open_tag` setting in the `php.ini` configuration file to `off`.

`DOCTYPE` After the XML declaration comes the `DOCTYPE` declaration. This particular `DOCTYPE` specifies that the root tag in the docuent will be `html`, that the document type is `PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"`, and that a DTD (Document Type Definition) for this document can be found at http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd.

DTD A DTD file describes the structure of a document type. A validating parser can use the DTD to determine whether a document conforms to the definition. Not all XML parsers are validating parsers; some check only whether a document is *well-formed*. A *well-formed* document complies with the basic XML standard, meaning that all elements in the document follow the XML standard. A *valid XML document*, on the other hand, conforms to the XML standard, and the DTD associated with the document type.

To check whether an XHTML or HTML document is valid according to a specified DOCTYPE, use the validator at `http://validator.w3c.org`.

The `root` element (`root node`) After the XML declaration and the DOCTYPE declaration, the remainder of the document is the content itself. The document begins with the `root node`:

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

The XHTML 1.0 Transitional DTD specifies that the root element (html) must contain an xmlns declaration for the XHTML namespace. A *namespace* provides a means of mixing two separate document types into one XML document, such as embedding WML or MathML into XHTML.

XML Namespaces Although namespaces look like a normal URL, they are not addresses of actual documents. A namespace is required within XML document so that XML parsers can determine which definitions to use for like-named elements. For example, the XHTML namespace has a `<title></title>` tagset, and the DOCBOOK namespace at `http://www.oasis-open.org/docbook/xml/simple/4.1.2.5/sdocbook.dtd` also has a `<title></title>` tagset. XML uses namespaces to apply the title tagset correctly. The example below illustrates the use of multiple namespaces within a single XML document. Notice the use of aliases formed by `xmlns:db="..."` for the Docbook DTD and `xmlns:h="..."` for the HTML DTD.

Example 2

```
<h:html xmlns:db="http://www.oasis-open.org/docbook/xml/simple/4.1.2.5/sdocbook.dtd"
        xmlns:h="http://www.w3.org/HTML/1998/html4">
 <h:head><h:title>Book Review</h:title></h:head>
 <h:body background="bg.png">
  <db:bookreview>
   <db:title>XML: A Primer</db:title>
   <h:table>
    <h:tr align="center">
     <h:td>Author</h:td><h:td>Price</h:td>
     <h:td>Pages</h:td><h:td>Date</h:td></h:tr>
    <h:tr align="left">
     <h:td><db:author>Simon St. Laurent</db:author></h:td>
     <h:td><db:price>31.98</db:price></h:td>
     <h:td><db:pages>352</db:pages></h:td>
     <h:td><db:date>1998/01</db:date></h:td>
    </h:tr>
   </h:table>
  </db:bookreview>
 </h:body>
</h:html>
```

The `h:head` tags contain the `h:title` tags and correspond to values in the HTML DTD. The `db:title` tags refer the formatting specification in the Docbook DTD and should not be confused with the HTML `title` tags.

Attributes The **h:body tag** include the background attribute. **Attributes** contain extra information about a specific tag. XML standards require all attributes to have a value enclosed in single or double quotes. It seems like a good idea to adhere to a consistent style of quoting. In this example, **background** specifies an image to be found in the file bg.png.

Another example of a correct attiribute is `<option selected="true"></option>`. Using the common `<option selected></option>` will generate an error because the `selected` attribute has no value.

Single-element Tags All opening tags, such as , need a matching closing tag, such as . Tags that in the past have stood alone, like the
 and <hr> tags have a special format. Rather than writing
</br>, we can write
 and <hr />. There must be a space before the closing /> in these single element tags.

Special Characters

Some characters are special in XML and cause problems when they are present in the content of a document. For example, the < and > characters are used for tags. If you use these characters in an XML document, they will be treated as part of a tag.

Entities Entities were developed so that users could insert special characters into a document without confusing XML. Entities are character combinations that begin with an ampersand (&) and end with a semi-color (;), that we can use in the content of our XML document instead of special characters. The entity will be recognized correctly and not treated as a special character. < will replace the < character, and > will replace the > character. XML handles entities corrrectly in your document and they will not be treated as tags. Entities are used to input non-ASCII characters into your XML document. The ë and € characters are represented by &euml and €.

If you want to use the & character itself, you must use the & entity.

3 Parsing XML

There are two methods for parsing XML documents in PHP: **SAX** (Simple API for XML) and **DOM** (Document Object Model). When PHP uses SAX, the parser goes through your XML document and detects events for every start and stop tag or other element found. You decide how to deal with the events the parser finds. When using DOM, the entire XML document is parsed into a tree that you can walk through using functions from PHP.

PHP 5 provides another way of parsing XML: the SimpleXML extentions. Before we discuss SimpleXML, we will take a look at the two mainstream methods.

3.1 SAX

PHP has built-in functions to handle parsing a file with SAX. We first create an XML parser object:

```
$xml = xml_parser_create('UTF-8');
```

The optional parameter UTF-8 denotes the encoding to use while parsing the XML document. If this function is successful, it return an XML parser handle that we will use with all the other XML parsing functions.

SAX works by handling events, beginning and ending, and we need to write the handlers ourselves. In this example we will focus on the two most important handlers that handle the start and end tag events, and the handler that deals with the character data (content).

```
// set handlers
xml_set_element_handler($xml, 'start_handler', 'end_handler');
xml_set_character_data_handler($xml, 'char_data_handler');
```

These statements establish the names of the start/end event handlers. The handler functions will be implemented in the following examples.

```
$level = 0;
// implement handlers
function start_handler($xml, $tag, $attributes)
{
    global $level;

    print "\n".str_repeat('  ', $level).">>>$tag";
    foreach($attributes as $k => $v) {
        print " $k $v";
    }
    $level++;
}
```

The `start_handler()` function is passed three arguments: the XML parser object, the name of the tag, and an array containing the attributes, if any, for the tag. The tag name is passed to the function with all characters uppercased if case folding is enabled (default). You can turn off uppercasing by setting an option with the XML parser object:

```
xml_parser_set_option($xml, XML_OPTION_CASE_FOLDING, false);
```

Keep in mind that XML is case-sensitive.

The `end_handler()` is passed the XML parsing object and the tag name. It is not passed the attributes array.

```
function end_handler($xml, $tag)
{
    global $level;
    $level--;
    print str_repeat('  ', $level) . "<<<$tag";
}
```

Now that we have handlers for the start and end of an event(tag), we need to handle the character data contained between the tags. We will use the `word_wrap()` function to make sure that the output fits on our screen.

```
function char_data_handler($xml, $data)
{
    global $level;
    $data = split("\n", wordwrap($data, 76 - ($level * 2)));
    foreach($data as $line) {
        print str_repeat('  ', $level + 1)).$line."\n";
    }
}
```

Now we can start parsing our XML file (Example 1).

```
xml_parse($xml, file_get_contents('example1.xml');
```

The first lines of the output look like this:

```
>>>html xmlns http://www.w3.org//1999/xhtml xml:lang en lang en  ||
  |  |

 >>>head   ||
   |    |

  >>>title    |XML Example|
  <<<title   ||
   |  |
 <<<head  ||
  |  |

 >>>body background bg.png   ||
   |    |
```

This output is ungainly and shows the vertical bars in every case the chararcter data handler function is called. The output can be cleaned up considerably by putting all data in a buffer and only calling the character data handler when a tag closes or when another tag starts. This example cleans up the output:

```
<?php
  // initialize variables
  $level = 0;
```

```
  $char_data = '';

  // create the XML parser handler
  $xml = xml_parser_create('UTF-8');

  // set handlers
  xml_set_element_handler($xml, 'start_handler', 'end_handler');
  xml_set_character_data_handler($xml, 'char_data_handler');

  // parse the whole file in one run
  xml_parse($xml, file_get_contents('example1.xml'));

/*************** F u n c t i o n s ***************/
/*
 * flush collected data from the character handler
 */
function flush_data()
{
    global $level, $char_data;
    /* trim data and dump it when data exists */
    $char_data = trim($char_data);

    if(strlen($char_data) > 0 ) {
        print "\n";
        // wrap the text so it fit a 80 column terminal
        $data = split("\n", wordwrap($char_data, 76 - ($level*2)));
        foreach($data as $line) {
            print str_repeat('  ', ($level +1)) . "[".$line."]"."\n";
        }
    }
    /* clear the data in the buffer */
    $char_data = '';
}

/*
 * handler for start tags
 */
function start_handler($xml, $tag, $attributes)
{
    global $level;
    /* flush collected data from the character handler */
    flush_data();
    /* dump attributes as string */
    echo "\n". str_repeat('  ', $level). "$tag";
    foreach($attributes as $key => $value) {
        print " $key='$value'";
    }
    /* increase indentation level */
    $level++;
}

function end_handler($xml, $tag)
{
    global $level;
```

```
    /* flush collected data from character handler */
    flush_data();
    /* decrease indentation level */
    $level--;
    print "\n".str_repeat('  ', $level)."/$tag";
}

function char_data_handler($xml, $data)
{
    global $level, $char_data;
    /* add the character data to the buffer */
    $char_data .= ' '. $data;
}
?>
```

With these formatting changes, the output looks better:

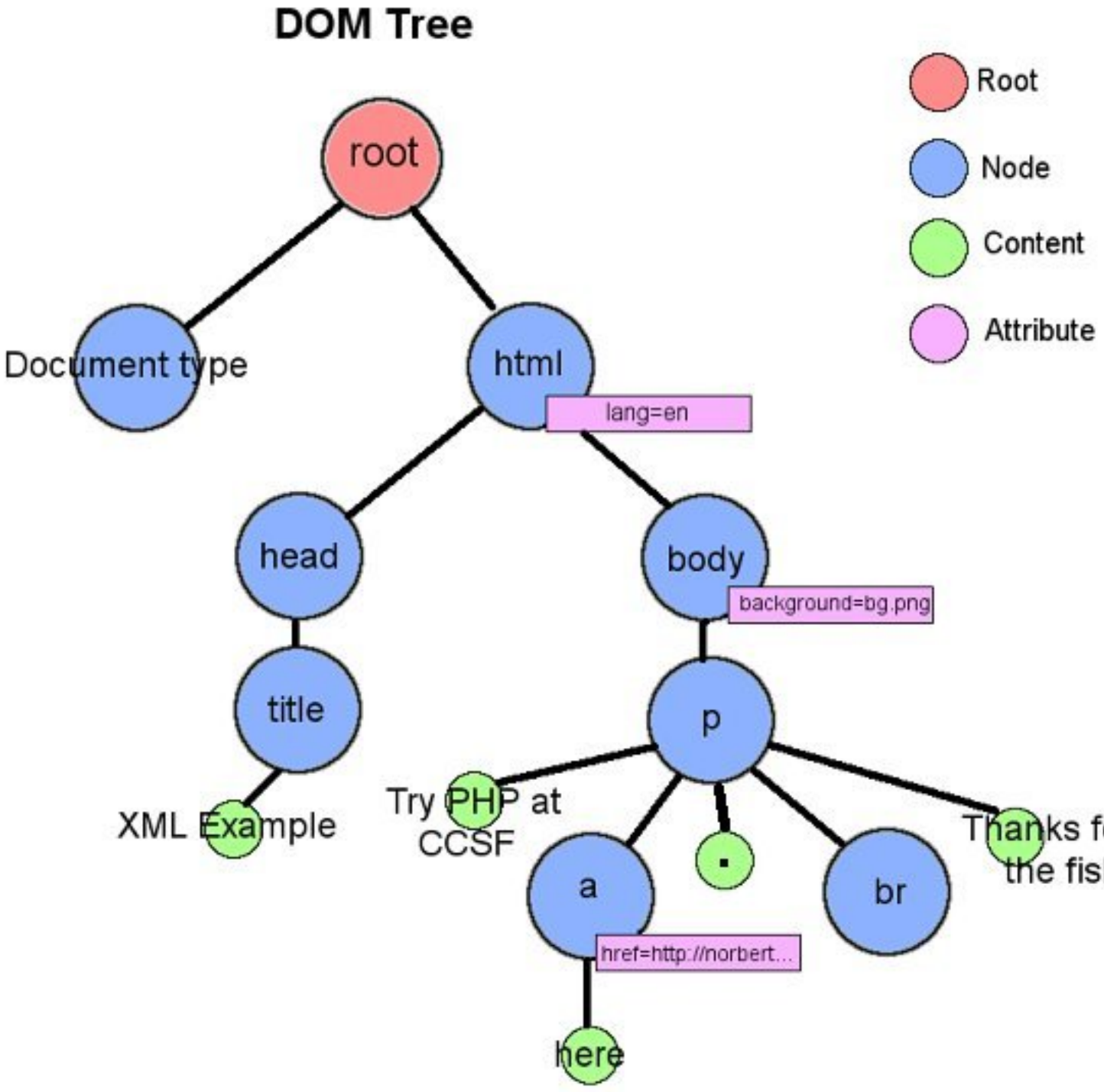
```
HTML XMLNS='http://www.w3.org//1999/xhtml' XML:LANG='en' LANG='en'
  HEAD
    TITLE
        [XML Example]

    /TITLE
  /HEAD
  BODY BACKGROUND='bg.png'
    P
        [Try Advanced PHP at CCSF]
      A HREF='http://norbert.ccsf.edu/cs130b'
          [here]

      /A
        [.]

      BR
      /BR
        [Thanks for all the fish!]

    /P
  /BODY
/HTML
```

3.2 DOM

As you see, parsing an XML file with SAX requires a lot of attention to detail. The DOM (`http://www.w3.org/TR/DOM-Level-3-Core/`) method makes the parsing job easier at the price of using much more system memory. For all relevant PHP documentation on the DOM refer to `http://www.php.net/manual/en/ref.dom`.

Rather than signaling the serial occurence of events in parsing the XML document, the DOM method creates a tree in memory of the entire XML file. If your XML file is many MB large, this process will consume large amounts of memory. The illustration below shows our example XML file displayed as a DOM tree.

DOM Tree



To display all of the content without tags, we can ”walk through” the tree of objects and pick out only the content. In the following example, we will go over all of the node children recursively.

```
1  <?php
2  $dom = new DomDocument();
3  $dom->load('example1.xml');
4  $root = $dom->documentElement;
```

```
 5
 6  process_children($root);
 7
 8  function process_children($node)
 9  {
10      $children = $node->childNodes;
11
12      foreach($children as $element) {
13          if ($element->nodeType == XML_TEXT_NODE) {
14              if (strlen(trim($element->nodeValue))) {
15                  print trim($element->nodeValue)."\n";
16              }
17          } elseif ($element->nodeType == XML_ELEMENT_NODE) {
18              process_children($element);
19          }
20      }
21  }
22  ?>
```

The output of this script looks like this:

```
XML Example
Try Advanced PHP at CCSF
here
.
Thanks for all the fish!
```

In this example, we read attributes of elements and do not call any methods. Line by line, it goes:

Line 4 Retrieve the root element of the DOM document that was loaded in Line 3

Lines 6, 18 Call `process_children()` when we encounter an element

Lines 13–16 If we find a text node, print its value

Lines 17–18 If we find an element, call `process_children()` recursively

We can look for specific attributes and print their values:

```
 1  <?php
 2  $dom = new DomDocument();
 3  $dom->load('example1.xml');
 4  $root = $dom->documentElement;
 5
 6  process_children($root);
 7
 8  function process_children($node)
 9  {
10      $children = $node->childNodes;
11
12      foreach($children as $element) {
13          if ($element->nodeType == XML_ELEMENT_NODE) {
14              if ($element->nodeName == 'body') {
15                    print $element->getAttributeNode('background')->value. "\n";
16                }
17                process_children($element);
```

```
18          }
19      }
20  }
21  ?>
```

The output of this recursive script is:

```
bg.png
```

Because we know the structure of the XML document, we can simplify the code considerably:

```
1  <?php
2    $dom = new DomDocument();
3    $dom->load('example1.xml');
4    $body = $dom->documentElement->getElementsByTagName('body')->item(0);
5    print $body->getAttributeNode('background')->value."\n";
6  ?>
```

Line 4 does most of the work in this script. It first request the `documentElement` of the XML document, the root node of the DOM tree. From that element, we request all child elements with the tag name body by user the getElementsByTagName. To get the value of the first body tag ($body) in the document—the one we know to be correct—we dereference `item[0]`. Once we have the specific tag we are interested in, we can get the background value by calling `getAttributeNode('background')` and display its value by reading the `value` property.

3.3 Using Xpath

XPath is a query language for XML documents, and is also used by XSLT for matching nodes. XPath can query a DOM document for certain nodes and attributes. Using XPath is similar to using SQL to query a database.

```
<?php
   $dom = new DomDocument();
   $dom->load('example1.xml');
   $xpath = new DomXpath($dom);
   $nodes = $xpath->query("*[local-name()='body']", $dom->documentElement);
   echo $nodes->item[0]->getAttributeNode('background')->value."\n";
?>
```

3.4 Creating a DOM Tree

In addition to parsing XML, the DOM extension can also create XML documents from scratch. A PHP script can build a tree of objects and dump them as an XML document. The following example will create a document similar to Example 1 above. The DOM extension is not concerned with making output that is human-friendly, so the whitespace in the output will be different that the example.

```
<?php
$dom = new DomDocument();

// create the root element
$html = $dom->createElement('html');

// create elements
$html->setAttribute("xmlns", "http://www.w3.org/1999/xhtml");
$html->setAttribute("xml:lang", "en");
$html->setAttribute("lang",  "en");

// add the html element to the document
$dom->appendChild($html);
```

We first create a `DomDocument` object, and procede to create elements with the `createElement()` and `createTextNode()` methods. The name of the element(`'html'`) is pass the the method, and a `DomElement` object is returned. Once we have a `DomElement`, we can add to it by calling the `appendChild()` method.

Next we add the the `title`, `body` and `background` attribute, and add the `<p>` element.

```
// add head element
$head = $dom->createElement('head');
$html->appendChild($head);

// add title element
$title = $dom->createElement('title');
$title->appendChild($dom->createTextNode("XML Example"));
$head->appendChild($title);

// create the body element
$body = $dom->createElement('body');
$body->setAttribute("background", "bg.png");
$html->appendChild($body);

// create the p element
$p = $dom->createElement('p');
$body->appendChild($p);
```

The `<p>` element is a more complicated than previous elements. It consists of: a text element ("`Try PHP at CCSF`"), an `<a>` element, another text element (the dot), the `<br>` element, and another text element ("`Thanks for all the fish!`")

```
// add the first text
$text = $dom->createTextNode("Try PHP at CCSF");
$p->appendChild($text);

// add the a
$a = $dom->createElement('a');
$a->setAttribute("href", "http://norbert.ccsf.edu/cs130b");
$a->appendChild($dom->createTextNode("here"));
$p->appendChild($a);

// add the "." and "Thanks for all the fish!"
$text = $dom->createTextNode(".");
$p->appendChild($a);

// add the <br>
$br = $dom->createElement('br');
$p->appendChild($br);

$text = $dom->createTextNode('Thanks for all the fish!');
$p->appendChild($text);
```

After this preparation, we can generate the XML document:

```
print $dom->saveXML();
?>
```

The OUTPUT:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"><head><title>XML
Example</title></head><body background="bg.png"><p>Try PHP at CCSF<a href="http:/
/norbert.ccsf.edu/cs130b">here</a><br/>Thanks for all the fish!</p></body></html>
```

3.5 SimpleXML

The SimpleXML extension is the easiest way to work with XML. There is no need to remember the DOM API, which is complex. With SimpleXML, you can access XML elements as if the document is a data structure. There are four rules:

1. Properties denote element iterators.
2. Numeric indices denote elements.
3. Non-numeric indices denote attributes.
4. String conversion allows access to TEXT data.

3.5.1 Creating a SimpleXML Document

There are three way to create a SimpleXML document:

```
<?php

// 1
$sxl = simplexml_load('example1.xml');

//2
$xml = <<<XML
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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" xml:lang="en" lang="en">
  <head>
    <title>XML Example</title>
  </head>
  <body background="bg.png">
    <p>
      Try  PHP at CCSF <a href="http://norbert.ccsf.edu/cs130b">here</a>.
    </p>
    <p>
      Check out the <a href="http://norbert.ccsf.edu/~somedude">Somedude Site</a>.
      <pre>
      Keep hacking...
      </pre>
    </p>
  </body>
</html>

XML;

// load the string into a SimpleXML document
$sx2 = simplexml_load_string($xml);

// 3
$sx3 = simplexml_load_dom($oDom);
?>
```

The first method, `simplexml_load_file()`, opens a file and parses it into memory. Method 2 uses `simplexml_load_string()` to parse the $xml string into memory. The third method, `simplexml_load_dom()`, imports a DOM object created with

DOM functions. The DOM extension has a corallary function named `dom_import_simplexml()` that allow us to share the same XML structures between both extensions.

3.5.2 Browsing SimpleXML Objects

Rule One states that *Properties denote element iterators*. This means that we can loop over all the `<p>` tags in the `<body>` element, like this:

```
<?php
foreach($sxl->body->p as $p) {
    // do something
}
?>
```

Rule Two states that *Numeric indices denote elements*. This means that we can access the second `<p>` tag with:

```
<?php
print $sxl->body->p[1];
?>
```

Rule Three tells us that *Non-numeric indices denote attributes*. This tells us that we can access the background attribute of the `body` tag like this:

```
<?php
print $sxl->body['background'];
?>
```

Rule Four say *String conversion allows access to TEXT data*. We can access all text data from the elements. In the following example, we can print the contents of the second `<p>` tag (combining rules 2 and 4):

```
<?php
print $sxl->body->p[1]
?>

OUTPUT:
Check out the
```

As you can see, accessing TEXT data does not include the text in the child nodes. We will use the `asXML()` method to access all of the text in the child nodes.

```
<?php
print $sxl->body->p[1]->asXML();
?>

OUTPUT:
    <p>
      Check out the <a href="http://norbert.ccsf.edu/~somedude">Somedude Site</a>.
      <pre>
      Keep hacking...
      </pre>
    </p>
```

To remove the tags we can apply `striptags()`:

```
<?php
print strip_tags($sx2->body->p[1]->asXML());
?>
```

```
OUTPUT:
Check out the Somedude Site.

Keep hacking...
```

In the following example we use SimpleXML to produce text from an XML document containing the information from the `vinyl` table. Here's a sample of the data:

```
<?xml version="1.0"?>
<vinyl>
    <row>
        <field name="vinyl_id">1</field>
        <field name="vinyl_sku">abc12345</field>
        <field name="vinyl_name">The Clash</field>
        <field name="vinyl_desc">The first release of THE ONLY BAND THAT MATTERS</field>
        <field name="vinyl_price">79.98</field>
        <field name="vinyl_qty">4</field>
        <field name="status">1</field>
        <field name="format">33</field>
        <field name="deleted">0</field>
        <field name="deleted_date">2005-08-14 17:11:06</field>
        <field name="created_date">2005-08-14 10:03:03</field>
        <field name="modified_date">2005-08-14 17:11:06</field>
        <field name="image_path">vinyl/sample.jpg</field>
        <field name="release_date">1977</field>
        <field name="condition">Near Mint</field>
        <field name="jacket_condition">Near Mint</field>
        <field name="reverse_image_path">vinyl/notavailable.jpg</field>
    </row>
    ...
</vinyl>
<?php
xdebug_disable();
/*
SimpleXMLElement Object
(
    [field] => Array
        (
            [0] => 1
            [1] => abc12345
            [2] => The Clash
            [3] => The first release of THE ONLY BAND THAT MATTERS
            [4] => 79.98
            [5] => 4
            [6] => 1
            [7] => 33
            [8] => 0
            [9] => 2005-08-14 17:11:06
            [10] => 2005-08-14 10:03:03
            [11] => 2005-08-14 17:11:06
            [12] => vinyl/sample.jpg
            [13] => 1977
            [14] => Near Mint
            [15] => Near Mint
```

```
            [16] => vinyl/notavailable.jpg
        )
)
*/
$sx1 = simplexml_load_file('vinyl.xml') or die();
// 1
foreach($sx1 as $row => $data) {
    print "Title    : ".$data->field[2]. "\n";
    print "Price    : ".$data->field[4]. "\n";
    print "Condition: ".$data->field[14]. "\n";
    print str_repeat('-', 30) ."\n";
}
?>
OUTPUT:
Title    : The Clash
Price    : 79.98
Condition: Near Mint
------------------------------
Title    : Give 'Em Enough Rope
Price    : 99.98
Condition: Near Mint
------------------------------
Title    : Sandinista
Price    : 39.98
Condition: Good
------------------------------
....
Title    : Studio Tan
Price    : 37.99
Condition: Near Mint
------------------------------
Title    : 200 Motels
Price    : 13.99
Condition: Near Mint
------------------------------
```

4 RSS

RSS (RDF Site Summary, Really Simple Syndication) is commonly used to provide users with yet another way to keep uptodate with what is going on with a web site. RSS is an XML vocabulary to describe news items, which can then be integrated into your own web site. PHP.net has an RSS feed at `http://www.php.net/news.rss`. The full specifications for RSS are at `http://web.resource.org/rss/1.0/spec`, but an example will be better help. Here is part of an RSS file generated by `http://norbert.ccsf.edu/~newguy/index.php?a=rssfeed`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns="http://purl.org/rss/1.0/" >
  <channel rdf:about="http://norbert.ccsf.edu/~newguy/index.php">
   <title>CS 130B News Feed</title>
    <items>
      <rdf:Seq>
       <rdf:li resource="http://norbert.ccsf.edu/~newguy/index.php?p=news&amp;id=12" />
       <rdf:li resource="http://norbert.ccsf.edu/~newguy/index.php?p=news&amp;id=14" />
```

```
        <rdf:li resource="http://norbert.ccsf.edu/~newguy/index.php?p=news&amp;id=13" />
       </rdf:Seq>
     </items>
   </channel>
   <item rdf:about="http://norbert.ccsf.edu/~newguy/index.php?p=news&amp;id=12">
     <title>Lecture 12 -- XML, DOM, and RSS</title>
     <link>http://norbert.ccsf.edu/~newguy/index.php?p=news&amp;id=12</link>
     <description>&lt;br /&gt;
LECTURE 12&lt;br /&gt;
&lt;br /&gt;
Lecture 12 is now available online. This week we are discussi...</description>
   </item>
   ... snip ...
</rdf:RDF>
```

RSS files consist of two parts: the header, describing the site from which the content is syndicated, and a list of available items. The second part consists of the items themselves. Usually RSS files are cached because they are not updated often. Downloading files once a day would be enough to keep current with most sites.

To read RSS files, we will install `XML_RSS` from `pear.php.net` by using `pear install XML_RSS`.

Example Script (rss_reader):

```
<style type=text/css>p {font-family:courier;}</style>

<h1>RSS Reader</h1>
This RSS Reader uses XML_RSS to format an RSS feed.
<?php
$sCacheFile = '/tmp/newguy.rss';
if(file_exists($sCacheFile) && filemtime($sCacheFile) < time() - 86400) {
    copy('http://norbert.ccsf.edu/~newguy/index.php?a=rssfeed', $sCacheFile);
}

require_once('XML/RSS.php');
$oRss = new XML_RSS($sCacheFile);

// parse the XML document into a data structure that can be fetched by other methods
$oRss->parse();

// use the getChannel() method
$aChannelInfo = $oRss->getChannelInfo();

print '<h1>'.$aChannelInfo['title'].'</h1>';

foreach($oRss->getItems() as $a) {
    print '<p><b>'. strtoupper($a['t:itle']).'</b><br>'."\n";
    print wordwrap($a['description'], 76, "<br />\n");
    print '<br><a href='.$a['link'].'>'.$a['link'].'</a></p>';
}
?>
```