

THE EXPERT'S VOICE® IN OPEN SOURCE

Covers PHP  
versions 5 and the  
forthcoming 6

# Pro PHP XML and Web Services

*Master working with XML and Web services using PHP*

Robert Richards

apress®



# XMLReader

**X**MLReader is a new stream-based parser in the PHP 5 lineup. If you skipped the previous chapter on the `xml` extension and do not know what a stream-based parser is, it may be beneficial to at least review that chapter because it explains in more detail what a stream-based parser is and how it works.

This chapter will introduce you to the XMLReader extension, explain the reasons for the existence of yet another stream-based parser, and show how to use this extension. The chapter will show how to use the API through short examples, with a complete example toward the end of this chapter. You can find additional examples of using this API in other chapters of this book, such as Chapter 14, which covers RDF, and Chapter 17, which covers REST. By the end of this chapter, you should understand what XMLReader is, know what its advantages and disadvantages are, and have a working knowledge of how to use the API in your everyday coding.

---

**Caution** Constants have been moved to class constants in PHP 5.1. This differs from PECL version 1.0.1 where constants are regular constants. The examples in this book use class constants to maintain compatibility with the PHP releases.

---

## Introducing XMLReader

The XMLReader extension is an object-oriented API that uses the `libxml2` implementation, which in turn is based on the C# implementation of the `XmlTextReader` API (<http://dotgnu.org/pnetlib-doc/System.Xml/XmlTextReader.html>). The XMLReader extension is a forward-only, stream-based parser, but unlike SAX, it is a pull rather than a push parser. As you move through a document, the parser's cursor positions itself on the different nodes, allowing you to access information from the current node. It offers many advantages over the `xml` extension, including additional functionality. As of PHP 5.1, this extension is part of the core PHP code base and can be built using the following configuration option:

```
--with-xmlreader
```

If you are still using PHP 5.0.x, the XMLReader extension is available from the PECL repository at <http://pecl.php.net/package/xmlReader>. You can install it using the PEAR installer or build it by adding it to your PHP source tree. Refer to the PHP manual for further information about building extensions.

## Push vs. Pull Parser

The previous chapter introduced you to stream-based parsing and the `xml` extension in particular. It explained that a *push* parser, in simple terms, pushes the data to your application while the XML is being parsed. The parser basically controls the flow of your application. A *pull* parser works much differently. It still operates on chunks of data at a time, providing a low memory footprint, but the application is in control of what data it wants and when the data is read from the stream. A pull parser allows you to free yourself from the control a push parser has over your application.

You can think of the difference between the two in terms of watching television. A push parser is like watching television without a digital video recorder. You are sitting there watching a show, and the commercials come on. If you are interested in anything in the commercials, you have to sit there and watch them, deciding which ones you like and which ones you don't. You can't get up and grab a snack, or you might miss something. You are not in control of the commercials. They, speaking in terms of a push parser, are pushed to your television, and you can't skip them, because you might want to watch one and can't pause them.

A pull parser, on the other hand, is like watching television with a digital video recorder (without the rewind feature, of course). By using the play, pause, and fast-forward buttons on the remote, you control the shows and commercials you watch. The current stream of XML data is comparable to the buffer of the digital video recorder. Like in the previous scenario, the commercials come on. Again, you might be interested in one of them. This time, you hit the pause button and grab something to eat. You return and decide you don't want to watch the commercial, so you fast-forward to the next one or even skip the next one. The push parser lets you control the movement of the parser, which is when data is read. When it stops at the point indicated, you can do anything you like in your code. The parser won't start reading more data until you tell it to do so. Your code could even stop reading the XML data and move on to something else. With a push parser, you really have no escape. Your application must read and act on everything in the buffer until all the data in the current stream has been read. It wouldn't be until the next `xml_parse()` call that you could safely stop reading XML data and have your application do something else.

This analogy may be a little over the top, but it should give you the idea. When using a pull parser, you are in control of the processing and of when data should be read. You are not at the mercy of the parser. As you will see in this chapter, this has many advantages over the traditional pull parser model, not to mention is much easier to use.

## Advantages Over the `xml` Extension

If you followed along with the previous section, comparing a push and pull parser to watching television without and with a digital video recorder, you may already have realized one of the advantages of `XMLReader` over the `xml` extension. With the `XMLReader` extension, you control when the data should be accessed. This is just one of the many benefits of this extension. Other advantages include better namespace support, streaming validation support, a simple API, and potentially faster processing.

### Namespace Support

From the examples in the previous chapter, you have most likely realized that processing namespaced documents is a real headache. The tag name is sent to the handler in the form

of the namespace URI, the separator, and the local name of the element concatenated together. It is up to you, as the developer, to split these based on the separator character just to get access to the name of the element. This doesn't even take into account what it requires to access the prefix, if any, of the element. The same goes for attributes.

Accessing namespace information is much simpler using XMLReader. Once the parser is positioned on an element, you can use the object properties to access both the URI and the prefix for the element. The API also allows access to both the local name and the qualified name of the element. No more jumping through hoops—the information is available in a simple-to-use manner. You can find more details and examples later in the “Dealing with Namespaces” section.

## Validation

If you want to perform validation, either by using RELAX NG or by using XML Schemas, you are pretty much out of luck with the xml extension. It will simply parse an XML document without regard to document validity. XMLReader adds the ability to validate a document while parsing. The API supports currently only RELAX NG, but with additions to the libxml2 library released in libxml2-2.6.20, PHP 5.2 should have XML Schema support. You can find further details and examples in the “Performing Validation” section.

## Simple API

The xml extension does not have an overly extensive or complicated API. You write handler functions and register them with the parser, and off it goes to do its thing. This sounds simple, right? Well it is, but unless you have used the extension before, XMLReader is much easier to use and understand. In fact, you can implement the majority of the xml extension's API using the XMLReader API with one method and two or three object properties. The XMLReader API is much larger than the xml API (though still very compact), but many of the properties and methods offer information not obtainable from the xml extension. In addition, XMLReader offers advanced functionality that is not available using the xml extension.

## Faster Processing

SAX parsing, which is what the xml extension does, should offer the fastest processing of XML data. Using PHP, however, this is typically not true. The numbers are close, but XMLReader can offer faster processing than the xml extension. Because you, as the developer, are in control of the parsing, data will be accessed only when needed. The xml extension, on the other hand, passes data around every time a registered event occurs. If the data from the event is not needed, it still is passed from the libxml2 library through the extension and finally to your handler. This data, to reach and be handled by the handler function, must also be converted into PHP usable data, such as the strings in PHP you already know.

XMLReader, on the other hand, allows you to move through the document, passing the minimal amount of data. When you reach a point of interest, you then request the specific data you want. Consider when the xml extension reaches the start of an element containing attributes. After testing the element tag name, the parser determines it can skip this element. The parser, though, has already processed all the attributes, packaged them, and sent them along as a parameter to the handler. With XMLReader, you request the attributes, so these are

not processed until you need or want them. It is small details like these that give XMLReader better performance. Just think of what you would need to do with namespaced documents.

Chapter 11 will return to this issue. Within that chapter, you'll compare all the different parser extensions in PHP 5, with respect to their speeds, using different methods and using different sets of data. You will find hard numbers comparing the processing speeds using the xml extension and using XMLReader, so you will be able to judge for yourself.

## Advanced Feature Set

Validation is one feature available using XMLReader and not the xml extension. Advanced namespace support in XMLReader is another. One of the best features of XMLReader, in my opinion, is the ability to be able to determine the type of node. Using the xml extension, in many cases the same handler handles different node types. For example, the character data handler takes care of text content and CDATA sections. You have no way to know what type of character data you are handling at the time. Text content is much different from CDATA, because CDATA can contain characters that are illegal to use as text content. This may affect how you need to handle the data.

Using the XMLReader API, you can easily access the type of node the parser is positioned on through a property from the object. Every type of node is available, so it is simple to process different types of data. A quick summary of some of the other features include the depth with the tree, the number of attributes held by an element, the exporting of nodes to the DOM extension, and the parser control while loading a document. This is just a subset of additional features, but these are some of the more important ones. Throughout this chapter, you will examine these features and see examples of them, so don't worry if you don't fully understand everything presented so far.

## Using XMLReader

XMLReader is an object-oriented API. If you couldn't tell by now, I happen to be a bit partial to OOP when dealing with XML APIs. I find it a bit more manageable to deal with documents in this manner. The steps you need to take to process a document are short and simple:

1. Create the XMLReader object.
2. Set any parser options not already set.
3. Parse the document.

## Creating the XMLReader Object

You can directly instantiate, or create, the XMLReader object using some methods statically. In this manner, it is similar to creating a DOMDocument object. The techniques aren't very different from each other. Calling the methods statically to create the object saves a line of code. You can directly instantiate the object in the same manner you normally create objects using the new keyword:

```
$objReader = new XMLReader();
```

The constructor takes no arguments and results in an object of type XMLReader.

This doesn't get you too far, though. The object is useless until it has a data stream. Data can be read from a string or directly from a file. This is an advantage over the SAX implementation. Using the `xml` extension, it is up to you to read the data from a file and then pass it to the parser. Here, the reader can take a URI and pull directly from it. The methods used in these cases are `open()` and `XML()`:

```
/* method prototypes */
boolean XMLReader::open(string URI)
boolean XMLReader::XML(string source)
```

The `open()` method is used to read data from a URI, which is specified by the URI parameter. The `XML()` method reads data from a string containing the document in memory, which is specified by the source parameter. Both methods return a Boolean indicating success or failure:

```
/* Set string data to read */
$data = '<root>my document</root>';
$objReader->XML($data);

/* Set URI pointing to document to parse */
$objReader->open('http://www.example.com/doc.xml');
```

You can also call these methods statically. This eliminates the need to first instantiate the `XMLReader` object:

```
/* Create object and set string data to read */
$data = '<root>my document</root>';
$objReader = XMLReader::XML($data);

/* Create object and set URI pointing to document to parse */
$objReader = XMLReader::open('http://www.example.com/doc.xml');
```

Creating the object first and then setting the input or doing it all at once using static methods is clearly up to you. You can save a few additional processing cycles by calling the methods statically, but unless this is critical to you, either way works just as well.

---

**Note** Throughout this chapter, the instantiated `XMLReader` object will simply be referred to as the *reader*.

---

One thing that is not currently possible using `XMLReader` but can be done using the `xml` extension is parsing an in-memory document that is broken up into multiple strings. For example, you can make multiple calls to `xml_parse()` where each call contains only a portion of the document to process. When using string data under `XMLReader`, the string must contain the entire document to be processed. This is something that may be expanded on in a future version, but for now when parsing large documents, using a file or stream and using the `open()` method are your best bets for keeping memory usage low. These do get processed via chunks of data without needing all the data to be residing in memory at one time.

## Setting Parser Properties

Once you have created the reader and set the input, you can set parser properties to further control how the document is parsed. You *must* set these properties after setting the input; otherwise, an error will be returned. XMLReader uses different parser options than the DOM and SimpleXML extensions, because of the libxml2 API. It was not possible to combine them into a single set of PHP constants. Table 9-1 describes the parser properties, which are basically a subset of the other libxml parser options from Chapter 5 for XMLReader.

---

**Caution** You must set parser properties *after* setting the input data on the XMLReader object. Any attempts to set these properties prior to setting the input will fail and either return FALSE or return an error message.

---

**Table 9-1.** XMLReader Parser Properties

Property	Value	Description
XMLREADER_DEFAULTATTRS	2	Forces the creation of default attributes within the document as defined in a DTD. With the current state of the XMLReader API in libxml2, default attributes are not available unless directly accessed by name. You can find a further explanation of this in the “Attributes” section later in this chapter.
XMLREADER_LOADDTD	1	Loads the DTD but does not validate the document.
XMLREADER_SUBST_ENTITIES	4	Substitutes entity references with their replaced content. Entity references will not be generated within the document.
XMLREADER_VALIDATE	3	Loads the DTD and validates the document based on the DTD while parsing.

You access parser properties through the `getParserProperty()` and `setParserProperty()` methods:

```
/* Method prototypes */
boolean XMLReader::getParserProperty(int property)
boolean XMLReader::setParserProperty(int property, boolean value)
```

The property parameter is one of the properties listed in Table 9-1. The value parameter for the `setParserProperty()` method is a Boolean indicating whether the specified property is enabled or disabled. The default value for all properties is FALSE. Both methods return a Boolean indicating whether the call succeeded or failed.

---

**Note** Some parser properties may not be changed after the initial read of the input data. For instance, a DTD may not be loaded after the reading of the data has already begun.

---

The following piece of code tests the value of the `XMLREADER_SUBST_ENTITIES` property. If the current value is `FALSE`, it then sets it to `TRUE`. It is a bit redundant and used only to illustrate both methods at once. When setting the value for a parser property, any existing value is overwritten, causing the `getParserProperty()` call in the following code snippet to be unnecessary:

```
if (! $objReader->getParserProperty(XMLREADER_SUBST_ENTITIES)) {
    $objReader->setParserProperty(XMLREADER_SUBST_ENTITIES, TRUE);
}
```

## Parsing the Document

Now that the reader is finally prepared, you can begin to parse the document. `XMLReader` is kind of a hybrid parser. The document is represented as nodes, just like with `DOM` and `SimpleXML`, but processed in a manner similar to the `xml` extension. Parsing the document consists of stopping at nodes along the way where the type of node encountered depends upon the method to position the parser. When performing a normal read with the reader, the parser will stop at all nodes except for attributes. You can access attributes differently than all other node types within a document. For this reason, attributes have their own section, “Attributes,” which deals the functionality available to deal with them. Table 9-2, for PHP 5.1 and higher, and Table 9-3, when using PECL version 1.0.1, describe the constants used for the node types you may encounter when using `XMLReader`.

**Table 9-2.** *XMLReader Node Type Constants*

Node Type	Value	Description
<code>XMLREADER::NONE</code>	0	No current node present. This type is encountered prior to the first read and after the entire document has been processed.
<code>XMLREADER::ELEMENT</code>	1	Element node. This type signals the starting tag of an element.
<code>XMLREADER::ATTRIBUTE</code>	2	Attribute node.
<code>XMLREADER::TEXT</code>	3	Text node.
<code>XMLREADER::CDATA</code>	4	CDATA section node.
<code>XMLREADER::ENTITY_REF</code>	5	Entity reference node.
<code>XMLREADER::ENTITY</code>	6	Entity node.
<code>XMLREADER::PI</code>	7	PI node.
<code>XMLREADER::COMMENT</code>	8	Comment node.
<code>XMLREADER::DOC</code>	9	Document node.
<code>XMLREADER::DOC_TYPE</code>	10	Document type node.
<code>XMLREADER::DOC_FRAGMENT</code>	11	Document fragment node.
<code>XMLREADER::NOTATION</code>	12	Notation node.
<code>XMLREADER::WHITESPACE</code>	13	Insignificant whitespace. This type of node is a result of being whitespace and within the scope of a node defining <code>xml:space</code> with the value of default.

*Continued*

Table 9-2. *Continued*

Node Type	Value	Description
XMLREADER::SIGNIFICANT_WHITESPACE	14	Significant whitespace. This is whitespace that either is being preserved from a node defining <code>xml:space</code> with the value of <code>preserve</code> or not in the scope of <code>xml:space</code> at all.
XMLREADER::END_ELEMENT	15	End element tag.
XMLREADER::END_ENTITY	16	End entity tag.
XMLREADER::XML_DECLARATION	17	XML declaration.

Table 9-3. *XMLReader Node Type Constants for PECL Version 1.0.1*

Node Type	Value	Description
XMLREADER_NONE	0	No current node present. This type is encountered prior to the first read and after the entire document has been processed.
XMLREADER_ELEMENT	1	Element node. This type signals the starting tag of an element.
XMLREADER_ATTRIBUTE	2	Attribute node.
XMLREADER_TEXT	3	Text node.
XMLREADER_CDATA	4	CDATA section node.
XMLREADER_ENTITY_REF	5	Entity reference node.
XMLREADER_ENTITY	6	Entity node.
XMLREADER_PI	7	PI node.
XMLREADER_COMMENT	8	Comment node.
XMLREADER_DOC	9	Document node.
XMLREADER_DOC_TYPE	10	Document type node.
XMLREADER_DOC_FRAGMENT	11	Document fragment node.
XMLREADER_NOTATION	12	Notation node.
XMLREADER_WHITESPACE	13	Insignificant whitespace. This type of node is a result of being whitespace and within the scope of a node defining <code>xml:space</code> with the value of <code>default</code> .
XMLREADER_SIGNIFICANT_WHITESPACE	14	Significant whitespace. This is whitespace that either is being preserved from a node defining <code>xml:space</code> with the value of <code>preserve</code> or is not in the scope of <code>xml:space</code> at all.
XMLREADER_END_ELEMENT	15	End element tag.
XMLREADER_END_ENTITY	16	End entity tag.
XMLREADER_XML_DECLARATION	17	XML declaration.

Not every node type listed in Table 9-2 is currently used. Some may be left over from older libxml2 code, and some may be for future use. For instance, it is doubtful that you will ever run into the node types `XMLREADER_ENTITY`, `XMLREADER_END_ENTITY`, and `XMLREADER_XML_DECLARATION`.

I will not say “never” here, because it is possible they may be used in a future version of libxml2; it is for this reason they are exposed through the XMLReader interface. It would be difficult to deal with node types that get implemented in the libxml2 API but are not exposed through the XMLReader extension, even though the constant has been available in older libxml2 versions.

I will use the document in Listing 9-1 within this chapter unless indicated otherwise. It represents the contents of an XML document within the file named `reader.xml`.

**Listing 9-1.** *Contents of File* `reader.xml`

```
<?xml version='1.0'?>
<!DOCTYPE chapter [
<!ELEMENT chapter (title, para, section)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT para ANY>
<!ATTLIST para name CDATA "default">
<!ELEMENT section ANY>
<!ATTLIST section id ID #REQUIRED>
]>
<chapter>
  <title>XMLReader</title>
  <para>
    First Paragraph
  </para>
  <section id="about">
    <title>About this Document</title>
    <para>
      <!-- this is a comment -->
      <?php echo 'Hi! This is PHP version ' . phpversion(); ?>
    </para>
  </section>
</chapter>
```

## Moving Through the Document

Unless you need access to attributes, moving through the document involves only two methods. These methods are `read()` and `next()`. In fact, you can access the entire document using only the `read()` method. Using the document from Listing 9-1, the reader will move to each node within the document and record the number of nodes accessed:

```
<?php
$objReader = XMLReader::open('reader.xml');
$count = 0;
while ($objReader->read()) {
  $count++;
}
print "Nodes Accessed: $count\n";
?>
```

The result of this example is the text `Nodes Accessed: 28`. Now I will explain what just happened and what 28 represents.

The `read()` method instructs the parser to move to the next node in the document, in document order. Once setting the input, think of the parser as being positioned on a document node. This is a concept from the tree parsers, but remember, `XMLReader` is a hybrid so the same concepts apply. Each time the `read()` method is called, the parser moves to the next node in the document, returning `TRUE` or `FALSE`. A return value of `FALSE` indicates that movement has failed, normally signaling that the end of the data stream has been reached. When using a well-formed document, this means the parser has reached the end of the document. Because of the construction of this method, you can access every node, except the attribute nodes, using the `read()` method within a `while` loop. Once the method returns `FALSE`, the end of the document has been reached, and execution moves to the next line of code following the end of the `while` block.

The initial `read` moves the cursor to the document type node. The XML declaration is skipped in this case. This is one of the cases where a node type is defined, `XMLREADER_XML_DECLARATION`, but it is not currently in use. No node types are available for the contents of the document type declaration, so the following `read` skips to the next node after it closes. If you are thinking that the next node encountered is an `XMLREADER_ELEMENT` node, representing the opening chapter tag, you are incorrect. The next node is actually the line breaks, which are `XMLREADER_SIGNIFICANT_WHITESPACE` nodes.

With this in mind, you can count the total number of nodes in the document. The number should total 28 because that is what the code indicated it would be:

- `XMLREADER_DOC_TYPE: 1`
- `XMLREADER_SIGNIFICANT_WHITESPACE: 11`
- `XMLREADER_ELEMENT: 6`
- `XMLREADER_END_ELEMENT: 6`
- `XMLREADER_TEXT: 2`
- `XMLREADER_COMMENT: 1`
- `XMLREADER_PI: 1`

And lo and behold, the total number of nodes in the document is 28. You might have come up with 29, but the line breaks within the first `para` element are actually part of the text content. Not to worry—I had to count a couple of times because my total kept coming out to 29.

The `next()` method is a little different from `read()`. It works on elements and moves the cursor much differently than the `read()` method does. Before trying to understand what it is exactly and how it works, it is necessary to understand the type of information available each time the cursor is positioned on a node. Once you understand how to use and access node information, you will revisit the `next()` method.

## Node Information

You access information for the current node through properties of the reader. All `XMLReader` properties are read-only. Remember, it's called *XMLReader* for a reason. Table 9-4 describes the properties and descriptions.

**Table 9-4.** XMLReader *Object Properties*

Property	Return Type	Description
attributeCount	int	The number of attributes when positioned on an element node. All other nodes return a value of 0.
baseURI	string	The base URI for the current node.
depth	int	The number of levels deep within the document tree. The depth begins at zero, so all nodes within the top-level scope of the document, such as the document element start and end tags, return a depth of 0.
hasAttributes	bool	A Boolean indicating the presence of attributes on the current node. This property will return FALSE for all node types other than XMLREADER_ELEMENT.
hasValue	bool	A Boolean indicating whether the current node, based on its type, can have a value. This does not mean that the current node actually has a value.
isDefault	bool	A Boolean indicating whether the attribute was generated from the default value in a DTD. Currently this property is not implemented in libxml2 and always returns FALSE.
isEmptyElement	bool	A Boolean indicating whether the current element is empty or FALSE in all other cases. An empty element is considered to be an empty-element tag only. <a /> will return TRUE, and <a></a> will return FALSE.
localName	string	The local name of the current node.
name	string	The qualified name of the current node.
namespaceURI	string	The namespace URI in which the current node resides.
nodeType	int	An integer representing a node type from Table 9-2 for the current node.
prefix	string	The prefix associated with the namespace for the current node.
value	string	The value for the current node or empty string when no value or node type cannot have a value.
xmlLang	string	The xml:lang in scope for the current node.

Comparing the difference between parsing with the xml extension and XMLReader clearly shows how much easier XMLReader is to use. The following code demonstrates what is involved to parse the reader.xml file and print element tags and character data:

```
<?php
function startElement($parser, $data, $attrs) {
    print "<".$data.">";
}

function endElement($parser, $data) {
    print $data;
}
```

```

function characterData($parser, $data) {
    print $data;
}

$xml_parser = xml_parser_create();
xml_parser_set_option ($xml_parser, XML_OPTION_CASE_FOLDING, 0);
xml_set_element_handler($xml_parser, "startElement", "endElement");
xml_set_character_data_handler($xml_parser, "characterData");
$handle = fopen("reader.xml", "r");
while ($data = fread($handle, 4096)) {
    if (!xml_parse($xml_parser, $data, feof($handle))) {
        break;
    }
}
fclose($handle);
?>

```

---

```

<chapter>
  <title>XMLReader</title>
  <para>
    First Paragraph
  </para>
  <section>
    <title>About this Document</title>
    <para>

    </para>
  </section>
</chapter>

```

---

You can get the same output using XMLReader, which not only is much easier to read but takes fewer lines of coding:

```

<?php
$objReader = XMLReader::open('reader.xml');
while ($objReader->read()) {
    switch ($objReader->nodeType) {
        case XMLREADER_ELEMENT:
            print "<".$objReader->localName.">";
            break;
        case XMLREADER_END_ELEMENT:
            print "</".$objReader->localName.">";
            break;
    }
}

```

```

        case XMLREADER_TEXT:
        case XMLREADER_CDATA:
        case XMLREADER_WHITESPACE:
        case XMLREADER_SIGNIFICANT_WHITESPACE:
            print $objReader->value;
    }
}
?>

```

Notice the last four case statements. `XMLReader` offers greater information for the data encountered in the document. While the reader sends all text and CDATA to the character data handler, each type of node, including whitespace, could be handled differently. In this case, you wanted the same behavior, so all text content is handled the same way. Try removing the whitespace types from the list of cases. The only line breaks in the output would be the line breaks that are part of the `First Paragraph` text node.

## The next() Method

When processing a document, it is not always the case that you need to access every single node. In fact, it is sometimes desirable to bypass an entire subtree and move to the next sibling node. The `next()` method provides this ability. When called, this method positions the cursor on the next node in the document, bypassing any subtree that may exist for the current node. This means only sibling nodes and nodes following the current node parent's starting tag will be accessed. For example:

```

<?php
$objReader = XMLReader::open('reader.xml');
/* Find the title element */
while ($objReader->read()) {
    if ($objReader->nodeType == XMLREADER_ELEMENT
        && $objReader->localName == "title") {
        break;
    }
}

/* find the section element that is a sibling of title */
while ($objReader->next()) {
    if ($objReader->nodeType == XMLREADER_ELEMENT
        && $objReader->localName == "section") {
        break;
    }
}

/* Descend into subtree of section element */
$objReader->read();
/* First whitespace node is skipped */

```

```

$depth = $objReader->depth;
while ($objReader->next()) {
    /* If depth is less than initial depth, cursor is out of the subtree */
    if ($objReader->depth < $depth) {
        print "\n**** Ascending rest of tree\n";
        print "Current Node: ".$objReader->localName;
        print " Type: ".$objReader->nodeType." Depth: ".$objReader->depth."\n";
        break;
    }
    print "Current Node: ".$objReader->localName;
    print " Type: ".$objReader->nodeType." Depth: ".$objReader->depth."\n";
}
?>

```

The code is a bit longer than it needs to be since the section node could have been initially searched for rather than the first title element node, but this example shows a couple ways of using the `next()` method.

The purpose of the first while block should be evident. The reader is moving to each node in the document until it encounters the first element start tag with the name `title`. Instead of using the `read()` method, the `next()` method is called, so from the title element node, the cursor moves to each sibling of this node until it encounters the section element node.

If you look at the document in Listing 9-1 again, you should notice the first child node for the section element is a significant whitespace. The cursor is positioned on this node using the `read()` method, but no processing or testing of the node is performed. Normally, unless you know the exact contents of the document being processed, this is not a good idea. For all you know, the document might not have any whitespaces, and the first child could be an important node type for the application. This is not the case here, so the lone call to the `read()` method is used to just move the cursor into the subtree of the section element.

The current depth within the document is now stored in the `$depth` variable, and the processing begins to see what nodes are actually encountered when calling `next()`. If you think about it, with the cursor positioned on the first text node (which is the significant whitespace), the siblings of this node are the title element, a text node that is whitespace, the para element, and another text node that is whitespace. Executing the code prints the following:

```

Current Node: title Type: 1 Depth: 2
Current Node: #text Type: 14 Depth: 2
Current Node: para Type: 1 Depth: 2
Current Node: #text Type: 14 Depth: 2

**** Ascending rest of tree
Current Node: section Type: 15 Depth: 1

```

The first four lines of output are exactly as expected: the two element nodes interspersed with significant whitespace nodes.

The next part of the output might throw you a bit. While accessing the sibling nodes, no end element nodes were encountered. When working with siblings, there is no need for the cursor to be positioned on the element end tag. The element nodes are encountered during the `next()` call, and positioning on the end tag would serve no purpose other than be a waste

of your time. When the end of a subtree has been reached, on the other hand, positioning back on the parent element node through its end tag can be useful. You may need to perform additional processing with the element based on some information obtained from its subtree. This explains why the last `next()` performed in the code results in the cursor being positioned on the end tag of the `section` element. Had processing not been stopped, the end tag for the `chapter` element would also have been reached.

This method also can take an optional parameter. You can supply the local name for the next node to position. The same rules apply using this parameter as when not using it, but the cursor will skip any nodes with a local name not matching the `localname` parameter. For instance, you could change the `while` loop that produced the previous output to stop only at the `para` element node:

```
while ($objReader->next("para")) {
    /* If depth is less than initial depth, cursor is out of the subtree */
    if ($objReader->depth < $depth) {
        print "\n**** Ascending rest of tree\n";
        print "Current Node: ".$objReader->localName;
        print " Type: ".$objReader->nodeType." Depth: ".$objReader->depth."\n";
        break;
    }
    print "Current Node: ".$objReader->localName;
    print " Type: ".$objReader->nodeType." Depth: ".$objReader->depth."\n";
}
```

---

```
Current Node: para Type: 1 Depth: 2
```

---

The `localname` parameter is not limited to elements. All node types have names, and these can be passed to the `next()` method as well. Try changing the `localname` parameter from `para` to `#text` in the `while` loop; your output should look like this:

```
Current Node: #text Type: 14 Depth: 2
Current Node: #text Type: 14 Depth: 2

**** Ascending rest of tree
Current Node: #text Type: 14 Depth: 1
```

## Accessing Attributes

You access attributes differently than all other nodes in a document. As you saw earlier in the “Moving Through the Document” section, `read()` did not stop on any attributes. Attributes are accessible only when positioned on an element node, with either the `XMLREADER_ELEMENT` node type or the `XMLREADER_END_ELEMENT` node type. From the list of properties, it is already evident that attributes exist and you can retrieve the number of attributes, but to physically access the attributes themselves involves using additional methods. You have two ways to retrieve information for attributes. You can retrieve attribute values while the cursor is positioned on an element, or you can move the cursor to specific attributes. The following subsections will use a different document to demonstrate the different methods.

```
$data = '<root att1="att1 value" att2="att2 value" att3="att3 value" />';
```

## Retrieving Attribute Values

You can retrieve attribute values using the `getAttribute()`, `getAttributeNo()`, and `getAttributeNS()` methods. I will discuss the latter method in the “Dealing with Namespaces” section. The difference between the remaining two methods is that `getAttribute()` takes a qualified name for its parameter while `getAttributeNo()` takes a zero-based index, identifying the position of the attribute in relative to the element, for its parameter:

```
$objReader = XMLReader::XML($data);
$objReader->read();
if ($objReader->nodeType == XMLREADER_ELEMENT && $objReader->hasAttributes) {
    print "att1: ".$objReader->getAttribute("att1")."\n";
    print "att2: ".$objReader->getAttribute("att2")."\n";
    print "att3: ".$objReader->getAttribute("att3")."\n";
    for ($x=0;$x < $objReader->attributeCount; $x++) {
        print "Attr Index $x: ".$objReader->getAttributeNo($x)."\n";
    }
}
```

---

```
att1: att1 value
att2: att2 value
att3: att3 value
Attr Index 0: att1 value
Attr Index 1: att2 value
Attr Index 2: att3 value
```

---

The results print the attribute value based on name. The last three lines of the results are from the `for` loop. The `for` loop executes its body one less time than the number of attributes on the element. The method `getAttributeNo()` works off a zero-based index, so the first attribute is at index 0. Each iteration through the loop prints the current attribute value based on the index `$x` and increments `$x` until it is equal to the number of attributes held on the element.

## Moving to Attributes

The problem with using the methods to retrieve attribute values from an element node is that you don't always know the attribute names. Or, the attributes live in namespaces, and you are unsure of the qualified names of the attributes. It is possible to get the values using the attribute index, but that still does not get you any closer to determining the name of the attribute or even whether the attribute lives in a namespace. The `XMLReader` API has a few methods that move the cursor to attribute nodes, which allows them to be accessed using object properties just like all other node types:

- `bool moveToAttribute(string qualifiedName):` Moves to an attribute by `qualifiedName`
- `bool moveToAttributeNo(int index):` Moves to an attribute by zero-based index
- `bool moveToAttributeNs(string localName, string namespaceURI):` Moves to an attribute with `localName` in the specified `namespaceURI`

- `bool moveToFirstAttribute()`: Moves to the first attribute in the list on the element
- `bool moveToNextAttribute()`: Moves to the next attribute in the list on the element

It is pretty obvious what these methods do based on their names. The following block of code will demonstrate how to use these methods, though I will demonstrate the method `moveToAttributeNs()` in the “Dealing with Namespaces” section.

One other method is handy when positioning the cursor on attributes. When positioned on an attribute, the method `moveToElement()` will position the cursor back on the element that owns the attribute. This allows the element to be accessed again. Otherwise, when positioned on an attribute, the method `read()` or `next()` is called, and the cursor moves as if the method were called while positioned on the element node for the attribute. For example:

```
$objReader = XMLReader::XML($data);
$objReader->read();
if ($objReader->nodeType == XMLREADER_ELEMENT && $objReader->hasAttributes) {
    $objReader->moveToAttribute("att1")."\n";
    print $objReader->localName.": ".$objReader->value."\n";

    $objReader->moveToAttributeNo(2)."\n";
    print $objReader->localName.": ".$objReader->value."\n";
    if ($objReader->moveToFirstAttribute()) {
        do {
            print $objReader->localName.": ".$objReader->value."\n";
        } while ($objReader->moveToNextAttribute());
    }

    $objReader->moveToElement();
    print $objReader->localName."\n";
}
```

---

```
att1: att1 value
att3: att3 value
att1: att1 value
att2: att2 value
att3: att3 value
root
```

---

Moving the cursor around attributes is similar to moving through the document, though you have much more freedom with attribute movement than with other types of nodes. Once the cursor is positioned on an attribute, the attribute node is accessible like every other node in the document. This is the only case, however, where the positioning can move in a reverse direction when working with `XMLReader`.

## Exporting to DOM Objects

XMLReader has a big advantage over the xml extension in that because of its internal API and being a hybrid parser, it is possible to export nodes to the DOM extension. You may ask why this is such a big deal. It may or may not be. It depends upon the functionality you need. Consider a few scenarios. You have a 100MB document, and you need to pull only a few nodes from it. You could need to create a new document based on these few nodes, or you could need to process these nodes using the XSL extension.

Of course, you could always load the document into the DOM or SimpleXML extension and access the nodes you need. This, however, will use more than 100MB of memory, because building a tree in memory will require much more memory than the size of the document. A better approach is to scan the document using XMLReader, export the specific nodes, and process them. This will keep memory to a minimum in this case. You can export by using the `expand()` method. The following example will use a small document to demonstrate this method:

```
$data = '<root><element att1="value">some text</element></root>';
$objReader = XMLReader::XML($data);
while($objReader->localName != "element") {
    $objReader->read();
}
if ($objReader->nodeType == XMLREADER_ELEMENT && $objReader->hasAttributes) {
    $objElement = $objReader->expand();
    var_dump($objElement);

    /* Use DOM API since these are DOM objects */
    $objAttribute = $objElement->attributes->item(0);
    print $objAttribute->nodeValue;
}
```

The cursor first moves to the node named `element`. Assuming the node is of `element` type and it has attributes, it is exported to a DOM object. To be precise, it is an `element` node, so it exports to an object of the `DOMElement` class, as shown by the `var_dump()`:

```
object(DOMElement)#2 (0) {
}
```

XMLReader is a stream-based parser, meaning that these nodes are not persistent. Exporting a node to DOM creates a copy (which is a real copy equivalent to cloning a node rather than the shared nodes passed between DOM and SimpleXML) of the XMLReader node that is not associated with any document. This is important. Without an associated document, the exported node is pretty much read-only.

## Dealing with Namespaces

Handling namespaces with XMLReader is not any harder than handling a document without namespaces. In fact, it works the same way with the same properties and methods you have been using all along. So why is this section dedicated to namespaces? The answer is simple.

It is easier to demonstrate how to work with namespaces after understanding the API rather than trying to understand everything at once.

The only real difference when working with namespaces is that a couple of methods and a few properties are relevant when dealing with namespaces but not otherwise.

---

**Tip** The next() method accepts a local name for its optional parameter. When working with prefixed elements, remember to not use the qualified name; just use the local name. The method getAttribute() will retrieve a namespaced attribute based on its local or qualified name, but remember from the XML specification that two attributes in different namespaces with the same local name may exist on the same element. Without using the qualified name and this method, you may not end up with the attribute value you intended to retrieve.

---

For the purposes of this chapter, I will use the document in Listing 9-2, referring to the file reader2.xml, as the basis for the XML data.

**Listing 9-2.** *Namespaced Document in File reader2.xml*

```
<?xml version='1.0'?>
<chapter xmlns:a="http://www.example.com/namespace-a"
        xmlns="http://www.example.com/default">
  <a:title>XMLReader</a:title>
  <para>
    First Paragraph
  </para>
  <a:section a:id="about">
    <title>About this Document</title>
    <para>
      <!-- this is a comment -->
      <?php echo 'Hi! This is PHP version ' . phpversion(); ?>
    </para>
  </a:section>
</chapter>
```

This document is basically the document from Listing 9-1 with the document type declaration removed, a default namespace (<http://www.example.com/default>) added, and an additional namespace (<http://www.example.com/namespace-a>) associated with the prefix a. A few of the elements and attributes have also been moved into the <http://www.example.com/namespace-a> namespace. Just to prove to you that namespaces do not alter the way nodes are accessed, I will run the original node count script again:

```

<?php
$objReader = XMLReader::open('reader2.xml');
$count = 0;
while ($objReader->read()) {
    $count++;
}
print "Nodes Accessed: $count\n";
?>

```

This time it outputs `Nodes Accessed: 27`. Now, don't go thinking I am trying to deceive you since the original one counted 28. The document type declaration has been removed, reducing the count by one. Other than that missing node, the cursor has stopped at the same nodes in this document as it did before.

## Prefixes and Namespace URIs

Let's take a look at some of the namespace-specific functionality. The first step is to position the cursor on the section element residing within the namespace prefixed by `a`:

```

$objReader = XMLReader::open('reader2.xml');
while ($objReader->read()) {
    if ($objReader->nodeType == XMLREADER_ELEMENT
        && $objReader->name == "a:section") {
        break;
    }
}
print $objReader->name;

```

Of course, this prints `a:section`; otherwise, this would have been futile. You could have also created the test for the node by doing this:

```

if ($objReader->nodeType == XMLREADER_ELEMENT
    && $objReader->localName == "section" && $objReader->prefix == "a") {
    break;
}

```

It is much easier using the qualified name in this case. Unlike a node not within a namespace or in the default namespace, the properties `localName` and `name` do not return the same thing for a node residing in a prefixed namespace. For example, when positioned on the `para` element, the following comparison is true:

```

/* This is TRUE for nodes in the default namespace or not residing in a namespace */
If ($objReader->name == $objReader->localName) {
    ...
}

```

Along with the `prefix` property, the `namespaceURI` property will return a string containing the namespace URI in which the node resides. Keep in mind the cursor is still positioned on the section element:

```

print $objReader->namespaceURI;

```

This prints `http://www.example.com/namespace-a`. As far as object properties go, `prefix` and `namespaceURI` are the only two that have meaning when dealing with namespaces and return empty strings in all other cases. The remaining properties, which you have already encountered, work the same way.

## Attributes

Attributes work pretty much in the same manner as explained previously. A few additional methods are specific to namespace usage as well as to the namespace declarations themselves. The first things to look at are the attribute methods. Two previously mentioned methods are `getAttributeNs()` and `moveToAttributeNs()`.

Both of these methods take two parameters. The first is the local name of the attribute, and the second is the `namespaceURI` in which the attribute is located. The section element, where the cursor is still positioned, has a single attribute with the local name `id` in the namespace `http://www.example.com/namespace-a`. You can retrieve the value of the attribute with any of the following calls:

```
print $objReader->getAttribute('id');
print $objReader->getAttribute('a:id');
print $objReader->getAttributeNs('id', 'http://www.example.com/namespace-a');
```

All three of these will print the value of the attribute named `id`. The first method is not recommended when working with namespaces. If an additional `id` attribute existed not within the same namespace, you have no guarantee which attribute value is being retrieved. Consider what might be printed if the start tag for the section element looked like `<a:section a:id="about" id="2">`. The value for the first attribute would be retrieved even though it was the second one you wanted.

---

**Caution** Do not use `getAttribute()` without qualified names unless trying to access a non-namespaced attribute. As of `libxml2 2.6.21`, this method will not retrieve values for namespaced attributes.

---

The `moveTo` methods work just like the `getAttribute` methods with regard to the qualified name. The `moveToAttribute()` method, however, does not have the bug the `getAttribute()` method has. When passing in a local name for the attribute, only non-namespaced attributes are retrieved:

```
$objReader->moveToAttribute('id');
print $objReader->value."\n";
$objReader->moveToAttribute('a:id');
print $objReader->value."\n";
$objReader->moveToAttributeNs('id', 'http://www.example.com/namespace-a');
print $objReader->value."\n";
```

Although you would expect the same results as using the `getAttribute` methods, it is slightly different:

```
/* first line is a blank line */
about
about
```

The output is actually correct. The `moveToAttribute()` method does not contain the bug in the `getAttribute()` method. In actuality, the previous results should have looked like these.

## Namespace Declarations

Namespace declarations are handled as regular attributes within `XMLReader`. They have their own section because the implementation is not complete in the `libxml2` library so can be accessed only from certain attribute methods. These methods are currently the `moveTo` methods, except the `moveToAttributeNs()` method. This method currently does not move the cursor to namespace declarations. For this example, the parser needs to be reset so the chapter element can be used:

```
<?php
$objReader = XMLReader::open('reader2.xml');
while ($objReader->read()) {
    if ($objReader->nodeType == XMLREADER_ELEMENT
        && $objReader->name == "chapter") {
        break;
    }
}

$objReader->moveToAttributeNo(0);
print $objReader->value."\n";
$objReader->moveToAttributeNo(1);
print $objReader->value."\n";

$objReader->moveToAttribute("xmlns:a");
print $objReader->value."\n";
$objReader->moveToAttribute("xmlns");
print $objReader->value."\n";

$objReader->moveToFirstAttribute();
print $objReader->value."\n";
$objReader->moveToNextAttribute();
print $objReader->value."\n";
?>
```

---

```
http://www.example.com/namespace-a
http://www.example.com/default
http://www.example.com/namespace-a
http://www.example.com/default
http://www.example.com/namespace-a
http://www.example.com/default
```

---

It is possible in the near future that additional attribute methods will support namespace declarations, but currently only the ones used previously in this chapter have been implemented as of libxml2-2.6.20.

## Performing Validation

One of the advantages over the xml extension is XMLReader's ability to perform validation while processing a document. Currently, only DTD and RELAX NG validation is supported, but by the time you read this, XML Schema support may have been added. Depending upon the type of validation being performed, you may need to prepare validation support before calling the initial `read()` method but after setting the input data stream. While processing the document, you can check the validity using the `isValid()` method. This method returns a Boolean indicating the state of document validity.

---

**Note** When not performing validation on a document, the `isValid()` method will always return `FALSE`.

---

## Validating with DTD

You specify validation using a DTD with the `XMLREADER_VALIDATE` parser property. When this property must be set depends upon a few conditions. When you need to load an external subset, you must set this property prior to the initial call to `read()` unless the `XMLREADER_LOADDTD` property has been set prior to the initial call to `read()`. By default an external subset is not loaded, so in order to ensure it is used, it must be loaded in order to validate the document. When the document does not contain an external subset, such as the document in Listing 9-1, you can set this property at any time during script execution. Until the `XMLREADER_VALIDATE` property has been set, however, any calls to `isValid()` will return `FALSE`, even though the document may be valid. Once the property has been set, `isValid` will begin to return the actual validity status of the document. For example:

```
<?php
$objReader = XMLReader::open('reader.xml');
$objReader->setParserProperty(XMLREADER_VALIDATE, TRUE);
while ($objReader->read()) {
    if (!$objReader->isValid()) {
        print "NOT VALID\n";
        break;
    }
}
?>
```

This piece of code results in no output. The only possible output would occur if the document were not valid at any time during processing.

## Validating with RELAX NG

RELAX NG validation works differently than DTD validation. The `isValid()` method is still used to check validity, but you instruct the reader to perform validation through the `setRelaxNGSchema()` method or the `setRelaxNGSchemaSource()` method. It is mandatory to call either method after setting the input data and prior to the first call to the `read()` method. Once the document has begun processing, the reader cannot be instructed to perform RELAX NG validation. For example:

```
<?php
$schema = '<?xml version="1.0" encoding="utf-8" ?>
<element name="chapter" xmlns="http://relaxng.org/ns/structure/1.0">
  <element name="title">
    <text/>
  </element>
  <element name="para">
    <text/>
  </element>
  <element name="section">
    <attribute name="id" />
    <text/>
  </element>
</element>';

$objReader = XMLReader::open('reader.xml');
$objReader->setRELAX_NGSchemaSource($schema);

libxml_use_internal_errors(TRUE);
while ($objReader->read()) {
  if (!$objReader->isValid()) {
    $xmlError = libxml_get_last_error();
    var_dump($xmlError);
    exit;
  }
}
?>
```

The schema defined by the `$schema` variable is used to validate the document from Listing 9-1 and is designed to fail. The reader is first instantiated, and the input data is set. With the reader prepared for parsing, the RELAX NG schema to validate against is set using the `setRelaxNGSchemaSource()` method, taking a string containing the entire schema as its parameter. For this example, the new error handling for XML, added in PHP 5.1, is used. This will allow the application to query for an XML error rather than having warnings displayed during script execution. Using the `read()` method, the reader moves throughout all the nodes in document order, checking the document validity at each stop, with `$objReader->isValid()`. Once the document fails validation, the script pulls the last error generated from the `libxml` library and dumps the structure to the output.

The section element from the schema is defined to allow only text content, but in the XML document itself, it actually contains child elements. Upon the reader encountering the child title element of the section element, the document fails the validity check, and the script prints the dump of the `LibXMLError` object obtained from the `libxml_get_last_error()` call:

```
object(LibXMLError)#2 (6) {
  ["level"]=>
  int(2)
  ["code"]=>
  int(38)
  ["column"]=>
  int(0)
  ["message"]=>
  string(35) "Did not expect element title there"
  ["file"]=>
  string(0) ""
  ["line"]=>
  int(0)
}
```

You can work with a RELAX NG schema from a file in the same manner. The only change would be to reference the schema as a file using `setRelaxNGSchema()`, passing the filename or URI as the parameter, rather than using a schema loaded into a string variable.

## Seeing Some Examples in Action

Throughout this chapter you have seen how XMLReader processes documents, but most examples have been small code snippets or code focusing on a particular functionality of the XMLReader API. It is time to look at a larger application that uses a good portion of the API and see how the code breaks down. For this example, just as in the previous chapter, I will process a document and build an in-memory tree. Although you could easily do this by exporting nodes using the `expand()` method, this example will not use that method; the node information will be processed using reader properties. The DOM extension will still be used to create the internal tree, but without relying on the `expand()` functionality, it is possible for you to easily implement your own tree creation storage by replacing the DOM functionality.

---

**Note** The complete example in this chapter presents the full API for the XMLReader extension in a single application. You can find real-world examples of using XMLReader in later chapters such as Chapter 14 and Chapter 17.

---

Here's the code:

```
<?php
class cReader extends XMLReader {
    private $document = NULL;
    private $currentNode = NULL;
    const xmlns = "http://www.w3.org/2000/xmlns/";

    public function __construct() {
        /* Create the base document for the tree */
        $this->document = new DOMDocument();
        $this->currentNode = $this->document;
    }

    function attributes() {
        /* DOM throws exceptions so try/catch used */
        try {
            if ($this->moveToFirstAttribute()) {
                do {
                    /* Attributes are always prefixed when in a namespace */
                    if ($this->prefix) {
                        if ($this->prefix != "xmlns") {
                            $this->currentNode->setAttributeNS($this->namespaceURI,
                                $this->name, $this->value);
                        } else {
                            /* This is a namespace declaration.
                               Ensure it is created as it may not be used on element */
                            $this->currentNode->setAttributeNS(self::xmlns,
                                $this->name, $this->value);
                        }
                    } else {
                        /* No need to handle default namespace declarations.
                           DOM already creates them with the element */
                        if ($this->name != "xmlns") {
                            $this->currentNode->setAttribute($this->name, $this->value);
                        }
                    }
                } while ($this->moveToNextAttribute());
            }
        } catch (DOMException $e) {
            throw $e;
        }
    }
}
```

```
function startElement() {
    try {
        if ($this->namespaceURI) {
            $node = $this->document->createElementNS($this->namespaceURI,
                $this->name);
        } else {
            $node = $this->document->createElement($this->name);
        }
        $this->currentNode = $this->currentNode->appendChild($node);
        if ($this->hasAttributes) {
            $this->attributes();
        }
    } catch (DOMException $e) {
        throw $e;
    }
}

function endElement() {
    $this->currentNode = $this->currentNode->parentNode;
}

function characterData() {
    try {
        $this->currentNode->appendChild(new DOMText($this->value));
    } catch (DOMException $e) {
        throw $e;
    }
}

function PIHandler() {
    $node = $this->document->createProcessingInstruction($this->name,
        $this->value);
    $this->currentNode->appendChild($node);
}

function saveXML() {
    return $this->document->saveXML();
}
}

$xmlldata = "<root><element1>text</element1><e2>text<e3>more</e3>text</e2></root>";

$objReader = new cReader();
$objReader->XML($xmlldata);
```

```

try {
    while ($objReader->read()) {
        switch ($objReader->nodeType) {
            case XMLREADER_ELEMENT:
                $objReader->startElement();
                break;
            case XMLREADER_END_ELEMENT:
                $objReader->endElement();
                break;
            case XMLREADER_TEXT:
            case XMLREADER_CDATA:
            case XMLREADER_WHITESPACE:
            case XMLREADER_SIGNIFICANT_WHITESPACE:
                $objReader->characterData();
                break;
            case XMLREADER_PI:
                $objReader->PIHandler();
                break;
        }
    }
} catch (DOMException $e) {
    var_dump($e);
}

print $objReader->saveXML();
?>

```

---

```

<?xml version="1.0"?>
<root><element1>text</element1><e2>text<e3>more</e3>text</e2></root>

```

---

This example performs the same functionality as shown in the example for the xml extension in Chapter 8. It is a bit longer because namespace support has been added. As you may infer from that, handling namespaces is much easier to deal with in XMLReader than in the xml extension. Let's take a look at the actual functionality contained in this example.

The cReader class is a class extending the XMLReader class. The only advantage of having written the functionality as object methods is that it is encapsulated and possibly a bit easier to follow. Before examining the class structure, let's jump right down to the actual body of the script itself where the cReader object is instantiated (again to be referred to as the *reader*) and then returned to the class itself.

The reader sets the input to the XML to process and is processed using the read() method. This ensures that the parser stops at each node within the document. The nodeType test mimics the behavior of the event handlers used in the previous chapter. This is the reason all the content type nodes—XMLREADER\_TEXT, XMLREADER\_CDATA, XMLREADER\_WHITESPACE, and XMLREADER\_SIGNIFICANT\_WHITESPACE—are grouped into the same functionality. The behavior

needs to be the same for all of these, although XMLReader-specific processing can be performed for each individual type. As each of the types listed in the `switch` statement are processed, the application calls the specified method from the object. This is similar to an event being called, but in this case the application controls the call; using the `xml` extension, it is called automatically.

The first type of node the document encounters is typically an element node. This is not always the case if you recall the possible legal structure of an XML document, but for this exercise, the document element will be the first node. When encountered, the `startElement()` method is called. The method first tests for a `namespaceURI` on the current node. When empty, a regular element is created; otherwise, the element resides in a namespace and is created as such. Notice that the element is created using the `name` property. This returns the qualified name of the node rather than using `localName`, which would return the name of the node without the appropriate prefix. The method then checks whether the element has attributes using the `hasAttributes` property. Remember that attributes are not a node type that the parser stops on. They must be requested when positioned on an element.

Assuming the element has attributes, the `attributes()` method is then called. This method may look a bit confusing. Support for namespace declarations has been added here because XMLReader handles namespace declarations just like any other attribute. The reader positions itself on the first attribute and begins the attribute processing. Attributes do not inherit the default namespace, so it is safe to assume that if an attribute has a prefix, the attribute is a namespaced attribute. Without the prefix, the attribute is handled as a normal attribute.

The case for a normal attribute also tests to make sure the attribute is not a default namespace declaration. In the case of this example, a default namespace would already have been created when the element was created. If you are unsure of the reason for this, refer to Chapter 6. Namespaced elements also need to test their prefixes for the string `"xmlns"`. These are also namespace declarations but define a prefix for the namespace as well. The DOM extension normally handles creating these when the element is created, but namespace declarations can also be defined on elements even though the element is not within the namespace. In a case like this, the namespace declaration simply needs to be created on the current element. Once the reader finishes with the current attribute, it moves to the next attribute using the `moveToNextAttribute()` method. This method is used as the truth expression for the `do/while` loop. This guarantees that the loop will be executed at least once, which is needed for the initial attribute, and will continue to be executed as long as the reader can move to the next attribute. The `TRUE/FALSE` return values from XMLReader methods make this extension extremely easy to use in control structures.

The remaining methods within the `cReader` class are fairly straightforward. The `name` and `value` properties retrieve the needed XML information. If you compare these methods to the equivalent ones from the previous chapter, you will find little difference other than how the XML information is passed and obtained. It also demonstrates how simple it can be to convert an existing application using the `xml` extension rather than using the XMLReader extension.

Again, you can customize this example if you like to use custom XML tree storage rather than the DOM extension. The DOM extension was used only for brevity, because it natively handles building an XML tree.

## Conclusion

This chapter introduced the XMLReader extension as well as many of the advantages it has over the xml extension. As of PHP 5.1, XMLReader has been included as part of the core PHP distribution but is also available from PECL for those running PHP 5.0.x. The explanations, code snippets, and examples in this chapter should provide you with enough information to immediately begin using this API. You can find additional real-world uses in Chapters 14 and 17. These may also help you understand some of the benefits of using XMLReader for XML processing.

XMLReader is the last of the native XML parsers in PHP. The next chapter will introduce you to XSLT and the XSL extension. You will begin to look at how you can transform XML data from one structure to another. If you have ever wondered how to use XML as a data source to produce many different types of output, such as HTML, XHTML, WAP, and so on, then the XSL extension is most likely what you have been seeking. Not only will you examine the extension, but you will learn how to write XSL templates as well.