NAME

XML::Writer – Perl extension for writing XML documents.

SYNOPSIS

```
use XML::Writer;
use IO::File;

my $output = IO::File->new(">output.xml");

my $writer = XML::Writer->new(OUTPUT => $output);
$writer->startTag("greeting",
                  "class" => "simple");
$writer->characters("Hello, world!");
$writer->endTag("greeting");
$writer->end();
$output->close();
```

DESCRIPTION

XML::Writer is a helper module for Perl programs that write an XML document. The module handles all escaping for attribute values and character data and constructs different types of markup, such as tags, comments, and processing instructions.

By default, the module performs several well-formedness checks to catch errors during output. This behaviour can be extremely useful during development and debugging, but it can be turned off for production-grade code.

The module can operate either in regular mode in or Namespace processing mode. In Namespace mode, the module will generate Namespace Declarations itself, and will perform additional checks on the output.

Additional support is available for a simplified data mode with no mixed content: newlines are automatically inserted around elements and elements can optionally be indented based as their nesting level.

METHODS

Writing XML

new([$params])

Create a new XML::Writer object:

```
my $writer = XML::Writer->new(OUTPUT => $output, NEWLINES => 1);
```

Arguments are an anonymous hash array of parameters:

OUTPUT

An object blessed into IO::Handle or one of its subclasses (such as IO::File), or a reference to a string, or any blessed object that has a *print()* method; if this parameter is not present, the module will write to standard output. If a string reference is passed, it will capture the generated XML (as a string; to get bytes use the `Encode` module).

If the string *self* is passed, the output will be captured internally by the object, and can be accessed via the `to_string()` method, or by calling the object in a string context.

```
my $writer = XML::Writer->new( OUTPUT => 'self' );

$writer->dataElement( hello => 'world' );

print $writer->to_string;  # outputs <hello>world</hello>
print "$writer";           # ditto
```

NAMESPACES

A true (1) or false (0, undef) value; if this parameter is present and its value is true, then the module will accept two-member array reference in the place of element and attribute names, as in

the following example:

```
my $rdfns = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
my $writer = XML::Writer->new(NAMESPACES => 1);
$writer->startTag([$rdfns, "Description"]);
```

The first member of the array is a namespace URI, and the second part is the local part of a qualified name. The module will automatically generate appropriate namespace declarations and will replace the URI part with a prefix.

PREFIX_MAP

A hash reference; if this parameter is present and the module is performing namespace processing (see the NAMESPACES parameter), then the module will use this hash to look up preferred prefixes for namespace URIs:

```
my $rdfns = "http://www.w3.org/1999/02/22-rdf-syntax-ns#";
my $writer = XML::Writer->new(NAMESPACES => 1,
                              PREFIX_MAP => {$rdfns => 'rdf'});
```

The keys in the hash table are namespace URIs, and the values are the associated prefixes. If there is not a preferred prefix for the namespace URI in this hash, then the module will automatically generate prefixes of the form "__NS1", "__NS2", etc.

To set the default namespace, use '' for the prefix.

FORCED_NS_DECLS

An array reference; if this parameter is present, the document element will contain declarations for all the given namespace URIs. Declaring namespaces in advance is particularly useful when a large number of elements from a namespace are siblings, but don't share a direct ancestor from the same namespace.

NEWLINES

A true or false value; if this parameter is present and its value is true, then the module will insert an extra newline before the closing delimiter of start, end, and empty tags to guarantee that the document does not end up as a single, long line. If the parameter is not present, the module will not insert the newlines.

UNSAFE

A true or false value; if this parameter is present and its value is true, then the module will skip most well-formedness error checking. If the parameter is not present, the module will perform the well-formedness error checking by default. Turn off error checking at your own risk!

DATA_MODE

A true or false value; if this parameter is present and its value is true, then the module will enter a special data mode, inserting newlines automatically around elements and (unless UNSAFE is also specified) reporting an error if any element has both characters and elements as content.

DATA_INDENT

A numeric value or white space; if this parameter is present, it represents the indent step for elements in data mode (it will be ignored when not in data mode). If it is white space it will be repeated for each level of indentation.

ENCODING

A character encoding to use for the output; currently this must be one of 'utf-8' or 'us-ascii'. If present, it will be used for the underlying character encoding and as the default in the XML declaration. All character data should be passed as Unicode strings when an encoding is set.

CHECK_PRINT

A true or false value; if this parameter is present and its value is true, all prints to the underlying output will be checked for success. Failures will cause a croak rather than being ignored.

end()

Finish creating an XML document. This method will check that the document has exactly one document element, and that all start tags are closed:

```
$writer->end();
```

If *OUTPUT* as been set to *self*, `end()` will return the generated document as well.

xmlDecl([$encoding, `$standalone`])

Add an XML declaration to the beginning of an XML document. The version will always be ‘‘1.0’’. If you provide a non-null encoding or standalone argument, its value will appear in the declaration (any non-null value for standalone except ’no’ will automatically be converted to ’yes’). If not given here, the encoding will be taken from the ENCODING argument. Pass the empty string to suppress this behaviour.

```
$writer->xmlDecl("UTF-8");
```

doctype($name, [$publicId, `$systemId`])

Add a DOCTYPE declaration to an XML document. The declaration must appear before the beginning of the root element. If you provide a publicId, you must provide a systemId as well, but you may provide just a system ID by passing ’undef’ for the publicId.

```
$writer->doctype("html");
```

comment($text)

Add a comment to an XML document. If the comment appears outside the document element (either before the first start tag or after the last end tag), the module will add a carriage return after it to improve readability. In data mode, comments will be treated as empty tags:

```
$writer->comment("This is a comment");
```

pi($target [, `$data`])

Add a processing instruction to an XML document:

```
$writer->pi('xml-stylesheet', 'href="style.css" type="text/css"');
```

If the processing instruction appears outside the document element (either before the first start tag or after the last end tag), the module will add a carriage return after it to improve readability.

The `$target` argument must be a single XML name. If you provide the `$data` argument, the module will insert its contents following the `$target` argument, separated by a single space.

startTag($name [, `$aname1` => `$value1`, ...])

Add a start tag to an XML document. Any arguments after the element name are assumed to be name/value pairs for attributes: the module will escape all ’&’, ’<’, ’>’, and ’"’ characters in the attribute values using the predefined XML entities:

```
$writer->startTag('doc', 'version' => '1.0',
                         'status' => 'draft',
                         'topic' => 'AT&T');
```

All start tags must eventually have matching end tags.

emptyTag($name [, `$aname1` => `$value1`, ...])

Add an empty tag to an XML document. Any arguments after the element name are assumed to be name/value pairs for attributes (see *startTag()* for details):

```
$writer->emptyTag('img', 'src' => 'portrait.jpg',
                         'alt' => 'Portrait of Emma.');
```

endTag([$name])

Add an end tag to an XML document. The end tag must match the closest open start tag, and there must be a matching and properly-nested end tag for every start tag:

```
$writer->endTag('doc');
```

If the $name argument is omitted, then the module will automatically supply the name of the currently open element:

```
$writer->startTag('p');
$writer->endTag();
```

dataElement($name, $data [, $aname1 => $value1, ...])
Print an entire element containing only character data. This is equivalent to

```
$writer->startTag($name [, $aname1 => $value1, ...]);
$writer->characters($data);
$writer->endTag($name);
```

characters($data)
Add character data to an XML document. All '<', '>', and '&' characters in the $data argument will automatically be escaped using the predefined XML entities:

```
$writer->characters("Here is the formula: ");
$writer->characters("a < 100 && a > 5");
```

You may invoke this method only within the document element (i.e. after the first start tag and before the last end tag).

In data mode, you must not use this method to add whitespace between elements.

raw($data)
Print data completely unquoted and unchecked to the XML document. For example `raw('<')` will print a literal < character. This necessarily bypasses all well-formedness checking, and is therefore only available in unsafe mode.

This can sometimes be useful for printing entities which are defined for your XML format but the module doesn't know about, for example   for XHTML.

cdata($data)
As `characters()` but writes the data quoted in a CDATA section, that is, between <![CDATA[and]]>. If the data to be written itself contains]]>, it will be written as several consecutive CDATA sections.

cdataElement($name, $data [, $aname1 => $value1, ...])
As `dataElement()` but the element content is written as one or more CDATA sections (see `cdata()`).

setOutput($output)
Set the current output destination, as in the OUTPUT parameter for the constructor.

getOutput()
Return the current output destination, as in the OUTPUT parameter for the constructor.

setDataMode($mode)
Enable or disable data mode, as in the DATA_MODE parameter for the constructor.

getDataMode()
Return the current data mode, as in the DATA_MODE parameter for the constructor.

setDataIndent($step)
Set the indent step for data mode, as in the DATA_INDENT parameter for the constructor.

getDataIndent()
Return the indent step for data mode, as in the DATA_INDENT parameter for the constructor.

Querying XML

in_element($name)
Return a true value if the most recent open element matches `$name`:

```
if ($writer->in_element('dl')) {
  $writer->startTag('dt');
} else {
  $writer->startTag('li');
}
```

within_element($name)
Return a true value if any open element matches `$name`:

```
if ($writer->within_element('body')) {
  $writer->startTag('h1');
} else {
  $writer->startTag('title');
}
```

current_element()
Return the name of the currently open element:

```
my $name = $writer->current_element();
```

This is the equivalent of

```
my $name = $writer->ancestor(0);
```

ancestor($n)
Return the name of the nth ancestor, where `$n`=0 for the current open element.

Additional Namespace Support
As of 0.510, these methods may be used while writing a document.

addPrefix($uri, `$prefix`)
Add a preferred mapping between a Namespace URI and a prefix. See also the PREFIX_MAP constructor parameter.

To set the default namespace, omit the `$prefix` parameter or set it to ''.

removePrefix($uri)
Remove a preferred mapping between a Namespace URI and a prefix.

forceNSDecl($uri)
Indicate that a namespace declaration for this URI should be included with the next element to be started.

ERROR REPORTING

With the default settings, the XML::Writer module can detect several basic XML well-formedness errors:

- Lack of a (top-level) document element, or multiple document elements.
- Unclosed start tags.
- Misplaced delimiters in the contents of processing instructions or comments.
- Misplaced or duplicate XML declaration(s).
- Misplaced or duplicate DOCTYPE declaration(s).
- Mismatch between the document type name in the DOCTYPE declaration and the name of the document element.
- Mismatched start and end tags.
- Attempts to insert character data outside the document element.
- Duplicate attributes with the same name.

During Namespace processing, the module can detect the following additional errors:

- Attempts to use PI targets or element or attribute names containing a colon.
- Attempts to use attributes with names beginning ''xmlns''.

To ensure full error detection, a program must also invoke the end method when it has finished writing a document:

```
$writer->startTag('greeting');
$writer->characters("Hello, world!");
$writer->endTag('greeting');
$writer->end();
```

This error reporting can catch many hidden bugs in Perl programs that create XML documents; however, if necessary, it can be turned off by providing an UNSAFE parameter:

```
my $writer = XML::Writer->new(OUTPUT => $output, UNSAFE => 1);
```

PRINTING OUTPUT

If *OUTPUT* has been set to *self* and the object has been called in a string context, it'll return the xml document.

to_string

If *OUTPUT* has been set to *self*, calls an implicit `end()` on the document and prints it. Dies if *OUTPUT* has been set to anything else.

AUTHOR

David Megginson <david@megginson.com>

COPYRIGHT AND LICENSE

SEE ALSO

XML::Parser