1 Writing a New Converter

Writing a New Converter — Overview of how to write a new Converter

Synopsis

and

Introduction to Converters

This documentation deals only with the advanced programming topic of creating a new output converter. To learn how to use phpDocumentor, read the {@tutorial phpDocumentor/phpDocumentor.pkg}. To learn how to extend an existing Converter, read {@tutorial Converters/Converters.pkg}

Basic Concepts

Abstract Parsing Data

Source Code Elements

A Converter's job is to take abstract data from parsing and create documentation. What is the abstract data? phpDocumentor is capable of documenting tutorials, and a php file and its re-usable or structurally important contents. In other words, phpDocumentor can document:

- A XML DocBook-based tutorial (see {@tutorial phpDocumentor/tutorials.pkg})
- Procedural Page (PHP source file)
- Include Statements
- Define Statements
- Global Variables
- Functions
- Classes
- Class Variables
- Class Methods

phpDocumentor represents these PHP elements using classes:

- {@link parserTutorial}
- {@link parserData, parserPage}
- {@link parserInclude}
- {@link parserDefine}
- {@link parserGlobal}
- {@link parserFunction}
- {@link parserClass}
- {@link parserVar}

- {@link parserMethod}

This relationship between the source code and abstract data is quite clear, and very simple. The data members of each abstract representation correspond with the information needed to display the documentation. All PHP elements contain a DocBlock, and this is represented by a class as well, {@link parserDocBlock}. The elements of a DocBlock are simple:

- Short Description, containing any number of inline tags
- Long Description, containing any number of inline tags
- Tags, some containing any number of inline tags in their general description field

phpDocumentor represents these elements using classes as well:

- {@link parserDesc} for both short and long descriptions
- {@link parserInlineTag} for inline tags
- {@link parserTag} for regular tags

HTML-specific issues

There are some other issues that Converters solve. In HTML, a link is represented by an <a> tag, but in the PDF Converter, it is represented by a <c:ilink> tag. How can we handle both cases? Through another abstract class, the {@link abstractLink} and its descendants:

- {@link tutorialLink}
- {@link pageLink}
- {@link defineLink}
- {@link globalLink}
- {@link functionLink}
- {@link classLink}
- {@link varLink}
- {@link methodLink}

Note the absence of an "includeLink" class - this is intentional, as only re-usable elements need to be linked. An include statement is always attached to the file that it is in, and cannot be anywhere else.

These abstract linking classes contain the information necessary to differentiate between the location of any of the element's documentation. They are only used to allow linking to documentation, and not to source code. A link is then converted to the appropriate text representation for the output format by {@link Converter::returnSee()} (a href=link for html, c:ilink:link for pdf, link linkend=link in xml:docbook, and so on).

The other issues solved by a converter involves html in a DocBlock. To allow better documentation, html is allowed in DocBlocks to format the output. Unfortunately, this complicates output in other formats. To solve this issue, phpDocumentor also parses out all allowed HTML (see {@tutorial phpDocumentor.howto.pkg#basics.desc} for more detailed information) into abstract structures:

- <b> -- emphasize/bold text
-
 -- hard line break, may be ignored by some converters
- <code> -- Use this to surround php code, some converters will highlight it
- <i> -- italicize/mark as important

- -- list item
- -- ordered list
- <p> -- If used to enclose all paragraphs, otherwise it will be considered text
- <pre> -- Preserve line breaks and spacing, and assume all tags are text (like XML's CDATA)
- -- unordered list

is mapped to classes:

- {@link parserB}
- {@link parserBr}
- {@link parserCode}
- {@link parserI}
- {@link parserList} - both types of lists are represented by this object, and each <li> is represented by an array item
- {@link parserPre}

<p> is represented by partitioning text into an array, where each array item is a new paragraph. (as in {@link parserDocBlock::$processed

With these simple structures and a few methods to handle them, the process of writing a new converter is straightforward.

Separation of data from display formatting

phpDocumentor has been designed to keep as much formatting out of the source code as possible. For many converters, there need be no new code written to support the conversion, as all output-specific information can be placed in template files. However, the complexity of generating class trees does require the insertion of some code into the source, so at the bare minimum, the {@tutorial Converter.methods.cls#override.getroottree} method must be overridden.

Methods that must be overridden

Creating a new converter can be challenging, but should not be too complicated. You need to override one data structure, {@link Converter::$leftindex}, to tell the Converter which of the individual element indexes your Converter will use. You will also need to override a few methods for the Converter to work. The most important relate to linking and output.

A Converter must override these core methods:

- {@tutorial Converter.methods.cls#core.convert} - take any descendant of parserElement or a parserPackagePage and convert it into output
- {@tutorial Converter.methods.cls#core.returnsee} - takes a abstract link and returns a string that links to an element's documentation
- {@tutorial Converter.methods.cls#core.returnlink} - takes a URL and text to display and returns an internet-enabled link
- {@tutorial Converter.methods.cls#core.output} - generate output, or perform other cleanup activities
- {@tutorial Converter.methods.cls#core.convert-ric} - Converts README/INSTALL/CHANGELOG file contents for inclusion in documentation
- {@tutorial Converter.methods.cls#core.converterrorlog} - formats errors and warnings from {@link $phpDocumentor_errors}. see {@link HTMLframesConverter::ConvertErrorLog()}
- {@tutorial Converter.methods.cls#core.getfunctionlink} - for all of the functions below, see {@link HTMLframesConverter::getFuncti for an example

- {@tutorial Converter.methods.cls#core.getclasslink}
- {@tutorial Converter.methods.cls#core.getdefinelink}
- {@tutorial Converter.methods.cls#core.getgloballink}
- {@tutorial Converter.methods.cls#core.getmethodlink}
- {@tutorial Converter.methods.cls#core.getvarlink}

A Converter may optionally implement these abstract methods:

- {@tutorial Converter.methods.cls#override.endpage} - do any post-processing of procedural page elements, possibly output documentation for the page
- {@tutorial Converter.methods.cls#override.endclass} - do any post-processing of class elements, possibly output documentation for the class
- {@tutorial Converter.methods.cls#override.formatindex} - format the {@link $elements} array into an index see {@link HTMLframesConverter::generateElementIndex()}
- {@tutorial Converter.methods.cls#override.formatpkgindex} - format the {@link $pkg_elements} array into an index see {@link HTMLframesConverter::generatePkgElementIndex()}
- {@tutorial Converter.methods.cls#override.formatleftindex} - format the {@link $elements} array into an index see {@link HTMLframesConverter::formatLeftIndex()}
- {@tutorial Converter.methods.cls#override.formattutorialtoc} - format the output of a {@}toc} tag, see {@link HTMLframesConverter::formatTutorialTOC()}
- {@tutorial Converter.methods.cls#override.getroottree} - generates an output-specific tree of class inheritance
- {@tutorial Converter.methods.cls#override.smartyinit} - initialize a {@link Smarty} template object
- {@tutorial Converter.methods.cls#override.writesource} - write out highlighted source code for a parsed file
- {@tutorial Converter.methods.cls#override.writeexample} - write out highlighted source code for an example
- {@tutorial Converter.methods.cls#override.unmangle} - do any post-processing of class elements, possibly output documentation for the class

The following methods may need to be overridden for proper functionality, and are for advanced situations.

- {@tutorial Converter.methods.cls#advanced.checkstate} - used by the {@link parserStringWithInlineTags::Convert()} cache to determine whether a cache hit or miss occurs
- {@tutorial Converter.methods.cls#advanced.getstate} - used by the {@link parserStringWithInlineTags::Convert()} cache to save state for the next Convert() call
- {@tutorial Converter.methods.cls#advanced.type-adjust} - used to enclose type names in proper tags as in {@link XMLDocBookConverter::type_adjust()}
- {@tutorial Converter.methods.cls#advanced.postprocess} - called on all converted text, so that any illegal characters may be escaped. The HTML Converters, for example, pass all output through {@link htmlentities()}
- {@tutorial Converter.methods.cls#advanced.gettutorialid} - called by the {@}id} inline tag to get the Converter's way of representing a document anchor

Converter methods an extended converter should use

- {@tutorial Converter.methods.cls#utility.getsortedclasstreefromclass} -- generating class trees by package
- {@tutorial Converter.methods.cls#utility.hastutorial} -- use this to retrieve a tutorial, or determine if it exists
- {@tutorial Converter.methods.cls#utility.gettutorialtree} -- use this to retrieve a hierarchical tree of tutorials that can be used to generate a table of contents for all tutorials
- {@tutorial Converter.methods.cls#utility.vardump-tree} -- use this to assist in debugging large tree structures of tutorials