NAME

Pod::Simple - framework for parsing Pod

SYNOPSIS

```
TODO
```

DESCRIPTION

Pod::Simple is a Perl library for parsing text in the Pod ("plain old documentation") markup language that is typically used for writing documentation for Perl and for Perl modules. The Pod format is explained *perlpod*; the most common formatter is called `perldoc`.

Pod formatters can use Pod::Simple to parse Pod documents and render them into plain text, HTML, or any number of other formats. Typically, such formatters will be subclasses of Pod::Simple, and so they will inherit its methods, like `parse_file`.

If you're reading this document just because you have a Pod-processing subclass that you want to use, this document (plus the documentation for the subclass) is probably all you need to read.

If you're reading this document because you want to write a formatter subclass, continue reading it and then read *Pod::Simple::Subclassing*, and then possibly even read *perlpodspec* (some of which is for parser-writers, but much of which is notes to formatter-writers).

MAIN METHODS

`$parser = `*`SomeClass`*`->new();`

This returns a new parser object, where *`SomeClass`* is a subclass of Pod::Simple.

`$parser->output_fh( *OUT );`

This sets the filehandle that `$parser`'s output will be written to. You can pass `*STDOUT`, otherwise you should probably do something like this:

```
my $outfile = "output.txt";
open TXTOUT, ">$outfile" or die "Can't write to $outfile: $!";
$parser->output_fh(*TXTOUT);
```

...before you call one of the `$parser->parse_`*`whatever`* methods.

`$parser->output_string( \$somestring );`

This sets the string that `$parser`'s output will be sent to, instead of any filehandle.

`$parser->parse_file( `*`$some_filename`*` );`

`$parser->parse_file( *INPUT_FH );`

This reads the Pod content of the file (or filehandle) that you specify, and processes it with that `$parser` object, according to however `$parser`'s class works, and according to whatever parser options you have set up for this `$parser` object.

`$parser->parse_string_document( `*`$all_content`*` );`

This works just like `parse_file` except that it reads the Pod content not from a file, but from a string that you have already in memory.

`$parser->parse_lines( `*`...@lines...`*`, undef );`

This processes the lines in `@lines` (where each list item must be a defined value, and must contain exactly one line of content -- so no items like `"foo\nbar"` are allowed). The final `undef` is used to indicate the end of document being parsed.

The other `parser_`*`whatever`* methods are meant to be called only once per `$parser` object; but `parse_lines` can be called as many times per `$parser` object as you want, as long as the last call (and only the last call) ends with an `undef` value.

`$parser->content_seen`

This returns true only if there has been any real content seen for this document. Returns false in cases where the document contains content, but does not make use of any Pod markup.

`*SomeClass*->filter( *$filename* );`

`*SomeClass*->filter( **INPUT_FH* );`

`*SomeClass*->filter( \*$document_content* );`

This is a shortcut method for creating a new parser object, setting the output handle to STDOUT, and then processing the specified file (or filehandle, or in-memory document). This is handy for one-liners like this:

```
  perl -MPod::Simple::Text -e
"Pod::Simple::Text->filter('thingy.pod')"
```

SECONDARY METHODS

Some of these methods might be of interest to general users, as well as of interest to formatter-writers.

Note that the general pattern here is that the accessor-methods read the attribute's value with `$value = $parser->*attribute*` and set the attribute's value with `$parser->*attribute*( *newvalue* )`. For each accessor, I typically only mention one syntax or another, based on which I think you are actually most likely to use.

`$parser->no_whining( *SOMEVALUE* )`

If you set this attribute to a true value, you will suppress the parser's complaints about irregularities in the Pod coding. By default, this attribute's value is false, meaning that irregularities will be reported.

Note that turning this attribute to true won't suppress one or two kinds of complaints about rarely occurring unrecoverable errors.

`$parser->no_errata_section( *SOMEVALUE* )`

If you set this attribute to a true value, you will stop the parser from generating a "POD ERRORS" section at the end of the document. By default, this attribute's value is false, meaning that an errata section will be generated, as necessary.

`$parser->complain_stderr( *SOMEVALUE* )`

If you set this attribute to a true value, it will send reports of parsing errors to STDERR. By default, this attribute's value is false, meaning that no output is sent to STDERR.

Setting `complain_stderr` also sets `no_errata_section`.

`$parser->source_filename`

This returns the filename that this parser object was set to read from.

`$parser->doc_has_started`

This returns true if `$parser` has read from a source, and has seen Pod content in it.

`$parser->source_dead`

This returns true if `$parser` has read from a source, and come to the end of that source.

`$parser->strip_verbatim_indent( *SOMEVALUE* )`

The perlpod spec for a Verbatim paragraph is "It should be reproduced exactly...", which means that the whitespace you've used to indent your verbatim blocks will be preserved in the output. This can be annoying for outputs such as HTML, where that whitespace will remain in front of every line. It's an unfortunate case where syntax is turned into semantics.

If the POD your parsing adheres to a consistent indentation policy, you can have such indentation stripped from the beginning of every line of your verbatim blocks. This method tells Pod::Simple what to strip. For two-space indents, you'd use:

```
  $parser->strip_verbatim_indent('  ');
```

For tab indents, you'd use a tab character:

```
  $parser->strip_verbatim_indent("\t");
```

If the POD is inconsistent about the indentation of verbatim blocks, but you have figured out a heuristic to determine how much a particular verbatim block is indented, you can pass a code reference instead. The code reference will be executed with one argument, an array reference of all the lines in the verbatim block, and should return the value to be stripped from each line. For example, if you decide that you're fine to use the first line of the verbatim block to set the standard for indentation of the rest of the block, you can look at the first line and return the appropriate value, like so:

```
  $new->strip_verbatim_indent(sub {
      my $lines = shift;
      (my $indent = $lines->[0]) =~ s/\S.*//;
      return $indent;
  });
```

If you'd rather treat each line individually, you can do that, too, by just transforming them in-place in the code reference and returning `undef`. Say that you don't want *any* lines indented. You can do something like this:

```
  $new->strip_verbatim_indent(sub {
      my $lines = shift;
      sub { s/^\s+// for @{ $lines },
      return undef;
  });
```

TERTIARY METHODS

`$parser->abandon_output_fh()`

Cancel output to the file handle. Any POD read by the `$parser` is not effected.

`$parser->abandon_output_string()`

Cancel output to the output string. Any POD read by the `$parser` is not effected.

`$parser->accept_code( @codes )`

Alias for *accept_codes*.

`$parser->accept_codes( @codes )`

Allows `$parser` to accept a list of *"Formatting Codes" in perlpod*. This can be used to implement user-defined codes.

`$parser->accept_directive_as_data( @directives )`

Allows `$parser` to accept a list of directives for data paragraphs. A directive is the label of a *"Command Paragraph" in perlpod*. A data paragraph is one delimited by `=begin/=for/=end` directives. This can be used to implement user-defined directives.

`$parser->accept_directive_as_processed( @directives )`

Allows `$parser` to accept a list of directives for processed paragraphs. A directive is the label of a *"Command Paragraph" in perlpod*. A processed paragraph is also known as *"Ordinary Paragraph" in perlpod*. This can be used to implement user-defined directives.

`$parser->accept_directive_as_verbatim( @directives )`

Allows `$parser` to accept a list of directives for *"Verbatim Paragraph" in perlpod*. A directive is the label of a *"Command Paragraph" in perlpod*. This can be used to implement user-defined directives.

`$parser->accept_target( @targets )`

Alias for *accept_targets*.

`$parser->accept_target_as_text( @targets )`

Alias for *accept_targets_as_text*.

`$parser->accept_targets( @targets )`

Accepts targets for `=begin/=for/=end` sections of the POD.

`$parser->accept_targets_as_text( @targets )`

Accepts targets for `=begin/=for/=end` sections that should be parsed as POD. For details, see *"About Data Paragraphs" in perlpodspec*.

`$parser->any_errata_seen()`

Used to check if any errata was seen.

Example:

```
  die "too many errors\n" if $parser->any_errata_seen();
```

`$parser->parse_from_file( $source, $to )`

Parses from `$source` file to `$to` file. Similar to *"parse_from_file" in Pod::Parser*.

`$parser->scream( @error_messages )`

Log an error that can't be ignored.

`$parser->unaccept_code( @codes )`

Alias for *unaccept_codes*.

`$parser->unaccept_codes( @codes )`

Removes `@codes` as valid codes for the parse.

`$parser->unaccept_directive( @directives )`

Alias for *unaccept_directives*.

`$parser->unaccept_directives( @directives )`

Removes `@directives` as valid directives for the parse.

`$parser->unaccept_target( @targets )`

Alias for *unaccept_targets*.

`$parser->unaccept_targets( @targets )`

Removes `@targets` as valid targets for the parse.

`$parser->version_report()`

Returns a string describing the version.

`$parser->whine( @error_messages )`

Log an error unless `$parser->no_whining( TRUE );`.

CAVEATS

This is just a beta release -- there are a good number of things still left to do. Notably, support for EBCDIC platforms is still half-done, an untested.

SEE ALSO

Pod::Simple::Subclassing

perlpod

perlpodspec

Pod::Escapes

perldoc

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, *http://github.com/theory/pod-simple/*. Feel free to fork and contribute, or to clone *git://github.com/theory/pod-simple.git* and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to <bug-pod-simple@rt.cpan.org>.

COPYRIGHT AND DISCLAIMERS

AUTHOR

Pod::Simple was created by Sean M. Burke <sburke@cpan.org>. But don't bother him, he's retired.

Pod::Simple is maintained by:

* Allison Randal `allison@perl.org`

* Hans Dieter Pearcey `hdp@cpan.org`

* David E. Wheeler `dwheeler@cpan.org`

Documentation has been contributed by:

* Gabor Szabo `szabgab@gmail.com`

* Shawn H Corey `SHCOREY at cpan.org`