PHP Style Guide

# Table Of Contents

Top of Form

1. Introduction 2

2. Conventions Used In This Document 2

3. Definitions 2

4. Basic Principles 4

5. The PHPDoc Format 5

5.1. Summary 5

5.2. Description 6

5.3. Tags 6

5.4. Inline PHPDoc 8

5.5. Examples 9

6. Inheritance 9

6.1. Class Or Interface 10

6.2. Function Or Method 11

6.3. Constant Or Property 11

7. Describing hashes 11

As @struct declaration 11

8. Tags 12

8.1. @api 12

8.2. @author 12

8.3. @category [deprecated] 13

8.4. @copyright 13

8.5. @deprecated 14

8.6. @example 14

8.7. @global 16

8.8. @internal 16

8.9. @license 17

8.10. @link [deprecated] 18

8.11. @method 19

8.12. @package 19

8.13. @param 20

8.14. @property 21

8.15. @return 21

8.16. @see 22

8.17. @since 23

8.18. @struct 23

8.19. @subpackage [deprecated] 23

8.20. @throws 24

8.21. @todo 25

8.22. @typedef 25

8.23. @uses 27

8.24. @var 27

8.25. @version 29

Appendix A. Types 29

ABNF 29

Details 30

Valid Class Name 31

Keyword 31

## 

## 1. Introduction

The main purpose of this PHP Standard Reference (PSR) is to provide a complete and formal definition of the PHPDoc standard. This PSR deviates from its predecessor, the de-facto PHPDoc Standard associated with [phpDocumentor 1.x](http://www.phpdoc.org/), to provide support for newer features in the PHP language and to address some of the shortcomings of its predecessor. This is an adaption of the PSR found in GITHUB at: <https://github.com/phpDocumentor>

This document SHALL NOT:

* Describe a standard for implementing annotations via PHPDoc. Although it does offer versatility which makes it possible to create a subsequent PSR based on current practices. See [chapter 5.3](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#53-tags) for more information on this topic.
* Describe best practices or recommendations for Coding Standards on the application of the PHPDoc standard. This document is limited to a formal specification of syntax and intention.

## 2. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](https://tools.ietf.org/html/rfc2119).

## 3. Definitions

* "PHPDoc" is a section of documentation which provides information on several aspects of a "Structural Element".

It is important to note that the PHPDoc and the DocBlock are two separate entities. The DocBlock is the combination of a DocComment, which is a type of comment, and a PHPDoc entity. It is the PHPDoc entity that contains the syntax as described in chapter 5 such as the description and tags.

* "Structural Element" is a collection of Programming Constructs which SHOULD be preceded by a DocBlock. The collection contains the following constructs:
  + file
  + require(\_once)
  + include(\_once)
  + class
  + interface
  + trait
  + function (including methods)
  + property
  + constant
  + variables, both local and global scope.

It is RECOMMENDED to precede a "Structural Element" with a DocBlock with its definition and not with each usage. It is common practice to have the DocBlock precede a Structural Element but it MAY also be separated by a an empty line.

Example:

/\*\* @var int $int This is a counter. \*/ $int = 0; // there should be no docblock here $int++;

or

/\*\* \* This class acts as an example on where to position a DocBlock. \*/ class Foo { /\*\* @var string|null $title contains a title for the Foo with a max. length of 24 characters \*/ protected $title = null; /\*\* \* Sets a single-line title. \* \* @param string $title A text with a maximum of 24 characters. \* \* @return void \*/ public function setTitle($title) { // there should be no docblock here $this->title = $title; } }

An example of use that falls beyond the scope of this Standard is to document the variable in a foreach explicitly; several IDEs use this information to assist their auto-completion functionality.

This Standard does not cover this specific instance as a foreach statement is not considered to be a "Structural Element" but a Control Flow statement.

/\*\* @var \Sqlite3 $sqlite \*/ foreach($connections as $sqlite) { // there should be no docblock here $sqlite->open('/my/database/path'); <...> }

* "DocComment" is a special type of comment which MUST
  + start with the character sequence /\*\* followed by a whitespace character
  + end with \*/ and
  + have zero or more lines in between.

In case a DocComment spans multiple lines then every line MUST start with an asterisk (\*) that SHOULD be aligned with the first asterisk of the opening clause.

Single line example:

/\*\* <...> \*/

Multiline example:

/\*\* \* <...> \*/

* "DocBlock" is a "DocComment" containing a single "PHPDoc" structure and represents the basic in-source representation.
* "Tag" is a single piece of meta information regarding a "Structural Element" or a component thereof.
* "Inline PHPDoc" is a "PHPDoc" that is related to a "Tag" instead of a "Structural element". It replaces the description part of the "Tag".
* "Type" is the determination of what type of data is associated with an element. This is commonly used when determining the exact values of arguments, constants, properties and more.

See Appendix A for more detailed information about types.

* "Semantic Version" refers to the definition as set in the [Semantic Versioning Specification 2.0.0](http://www.semver.org/).
* "FQSEN" is an abbreviation for Fully Qualified Structural Element Name. This notation expands on the Fully Qualified Class Name and adds a notation to identify class/interface/trait members and re-apply the principles of the FQCN to Interfaces, Traits, Functions and global Constants.

The following notations can be used per type of "Structural Element":

Namespace: \My\Space Function: \My\Space\myFunction() Constant: \My\Space\MY\_CONSTANT Class:\My\Space\MyClass Interface: \My\Space\MyInterface Trait: \My\Space\MyTrait Method:\My\Space\MyClass::myMethod() Property: \My\Space\MyClass::$my\_property Class Constant:\My\Space\MyClass::MY\_CONSTANT

A FQSEN has the following [ABNF](https://tools.ietf.org/html/rfc5234) definition:

FQSEN = fqnn / fqcn / constant / method / property / function fqnn = "\" [name] \*("\" [name]) fqcn = fqnn "\" name constant = (fqnn "\" / fqcn "::") name method = fqcn "::" name "()" property = fqcn "::$" name function = fqnn "\" name "()" name = (ALPHA / "\_") \*(ALPHA / DIGIT / "\_")

## 4. Basic Principles

* A PHPDoc MUST always be contained in a "DocComment"; the combination of these two is called a "DocBlock".
* A DocBlock MUST directly precede a "Structural Element"

An exception to this principle is the File-level DocBlock which MUST be placed at the top of a PHP source code file as the first DocBlock in a file.

To prevent ambiguity when a Structural Element comes directly after a File-level DocBlock MUST that element have its own DocBlock in addition to the File-level DocBlock.

Example of a valid File-level DocBlock:

<?php /\*\* \* This is a file-level DocBlock \*/ /\*\* \* This is a class DocBlock \*/ class MyClass { }

Example of an invalid File-level DocBlock

<?php /\*\* \* This is a class DocBlock \*/ class MyClass { }

## 5. The PHPDoc Format

The PHPDoc format has the following [ABNF](https://tools.ietf.org/html/rfc5234) definition:

PHPDoc = [summary] [description] [tags] inline-phpdoc = "{" \*SP PHPDoc \*SP "}" summary = \*CHAR ("." 1\*CRLF / 2\*CRLF) description = 1\*(CHAR / inline-tag) 1\*CRLF ; any amount of characters ; with inline tags inside tags = \*(tag 1\*CRLF) inline-tag = "{" tag "}" tag = "@" tag-name [":" tag-specialization] [tag-details] tag-name = (ALPHA / "\") \*(ALPHA / DIGIT / "\" / "-" / "\_") tag-specialization = 1\*(ALPHA / DIGIT / "-") tag-details = \*SP (SP tag-description / tag-signature / inline-phpdoc) tag-description = 1\*(CHAR / CRLF) tag-signature = "(" \*tag-argument ")" tag-argument = \*SP 1\*CHAR [","] \*SP

Examples of use are included in chapter 5.4.

### 5.1. Summary

A Summary MUST contain an abstract of the "Structural Element" defining the purpose. It is RECOMMENDED for Summaries to span a single line or at most two but not more than that.

A Summary MUST end with either

* a full stop (.) followed by a line break
* or two sequential line breaks.

If a Description is provided, then it MUST be preceded by a Summary. Otherwise the Description will be considered the Summary, until the end of the Summary is reached.

Because a Summary is comparable to a chapter title it is beneficial to use as little formatting as possible. As such, contrary to the Description (see next chapter), no recommendation is done to support a mark-up language. It is explicitly left up to the implementing application whether it wants to support this or not.

### 5.2. Description

The Description is OPTIONAL but SHOULD be included when the "Structural Element", which this DocBlock precedes, contains more operations, or more complex operations, than can be described in the Summary alone.

Any application parsing the Description is RECOMMENDED to support the Markdown mark-up language for this field so that it is possible for the author to provide formatting and a clear way of representing code examples.

Common uses for the Description are (amongst others):

* To provide more detail than the Summary on what this method does.
* To specify of what child elements an input or output array, or object, is composed.
* To provide a set of common use cases or scenarios in which the "Structural Element" may be applied.

### 5.3. Tags

Tags provide a way for authors to supply concise meta-data regarding the succeeding "Structural Element". Each tag starts on a new line, followed by an at-sign (@) and a tag-name followed by white-space and meta-data (including a description) or Inline PHPDoc.

If meta-data is provided, it MAY span multiple lines and COULD follow a strict format, and as such provide parameters, as dictated by the type of tag. The type of the tag can be derived from its name.

For example:

@param string $argument1 This is a parameter.

The above tag consists of a name ('param') and meta-data ('string $argument1 This is a parameter.') where the meta-data is split into a "Type" ('string'), variable name ('$argument') and description ('This is a parameter.').

The description of a tag MUST support Markdown as a formatting language. Due to the nature of Markdown it is legal to start the description of the tag on the same or the subsequent line and interpret it in the same way.

So the following tags are semantically identical:

/\*\* \* @var string This is a description. \* @var string This is a \* description. \* @var string \* This is a description. \*/

A variation of this is where, instead of a description, a tag signature is used; in most cases the tag will in fact be an "Annotation". The tag signature is able to provide the annotation with parameters regarding its operation.

If a tag signature is present then there MUST NOT be a description present in the same tag.

The meta-data supplied by tags could result in a change of actual runtime behavior of the succeeding "Structural Element", in which case the term "Annotation" is commonly used instead of "Tag".

Annotations will not be described in further detail in this specification as this falls beyond scope. This specification provides a basis on top of which annotations may be implemented.

#### 5.3.1. Tag Name

Tag names indicate what type of information is represented by this tag, or in case of annotations which behaviour must be injected into the succeeding "Structural Element".

In support of annotations, it is allowable to introduce a set of tags designed specifically for an individual application or subset of applications (and thus not covered by this specification).

These tags, or annotations, MUST provide a namespace by either

* prefixing the tag name with a PHP-style namespace, or by
* prefixing the tag name with a single vendor-name followed by a hyphen.

Example of a tag name prefixed with a php-style namespace (the prefixing slash is OPTIONAL):

@\Doctrine\Orm\Mapping\Entity()

Note: The PHPDoc Standard DOES NOT make assumptions on the meaning of a tag unless specified in this document or subsequent additions or extensions.

This means that you CAN use namespace aliases as long as a prefixing namespace element is provided. Thus the following is legal as well:

@Mapping\Entity()

Your own library or application may check for namespace aliases and make a FQCN from this; this has no impact on this standard.

Important: Tools using the PHPDoc Standard MAY interpret namespaces that are registered with that application and apply custom behaviour.

Example of a tag name prefixed with a vendor name and hyphen:

@phpdoc-event transformer.transform.pre

Tag names that are not prefixed with a vendor or namespace MUST be described in this specification (see chapter 7) and/or any official addendum.

#### 5.3.2. Tag Specialization

In order to provide a method by which to provide nuance to the tags defined in this standard but without expanding the base set, a tag specialization MAY be provided after the tag name by adding a colon followed by a string that provides a more nuanced description of the tag. The list of supported tag specializations is not maintained in this document as it may change over time. The meta document will contain a series of recommendations on a per-tag name basis, but projects are free to choose their own tag specializations if applicable.

Important: Tools using the PHPDoc Standard MAY interpret tag specializations that are registered with/understood by that application and apply custom behaviour, but are only expected to implement the preceding tag name as defined in this standard.

For example:

@see:unit-test \Mapping\EntityTest::testGetId

The above tag consists of a name ('see') and tag specialization ('unit-test'), and thus defines a relation to the unit test for the proceeding method.

#### 5.3.3. Tag Signature

Tag signatures are commonly used for annotations to supply additional meta-data specific to the current tag.

The supplied meta-data can influence the behavior of the owning annotation and as such influence the behavior of the succeeding "Structural Element".

The contents of a signature are to be determined by the tag type (as described in the tag-name) and fall beyond the scope of this specification. However, a tag-signature MUST NOT be followed by a description or other form of meta-data.

### 5.4. Inline PHPDoc

Specific Tags MAY have an "Inline PHPDoc" section at the end of the "Tag" definition. An "Inline PHPDoc" is a "PHPDoc" element enclosed in braces and is only present at the end of a "Tag" sequence, unless specified otherwise in a "Tag" definition. The "Inline PHPDoc" element MUST replace any description that COULD have been provided.

An example is the @method tag. This tag can be augmented using an "Inline PHPDoc" to provide additional information regarding the parameters, return value or any other tag supported by functions and methods.

An example of this is:

/\*\* \* @method integer MyMagicMethod(string $argument1) { \* This is the summary for MyMagicMethod. \* \* @param string $argument1 Description for argument 1. \* \* @return integer \* } \*/ class MyMagicClass { ... }

In this example is described how the @method tag for the Magic Method MyMagicMethod has a complete PHPDoc definition associated with it. In this definition all constraints, constructs and tags that apply to a normal usage of PHPDoc also apply.

The meaning of an "Inline PHPDoc" element differs based on the context in which it is provided. In the example above the "Inline PHPDoc" provides a regular PHPDoc definition as would precede a method.

To prevent confusion regarding the function of "Inline PHPDoc" elements MUST their usage be restricted to tags and locations that are documented.

### 5.5. Examples

The following examples serve to illustrate the basic use of DocBlocks; it is advised to read through the list of tags in chapter 8.

A complete example could look like the following example:

/\*\* \* This is a Summary. \* \* This is a Description. It may span multiple lines \* or contain 'code' examples using the \_Markdown\_ markup \* language. \* \* @see Markdown \* \* @param int $parameter1 A parameter description. \* @param \Exception $e Another parameter description. \* \* @\Doctrine\Orm\Mapper\Entity() \* \* @return string \*/ function test($parameter1, $e) { ... }

It is also allowed to omit the Description:

/\*\* \* This is a Summary. \* \* @see Markdown \* \* @param int $parameter1 A parameter description. \* @param \Exception $parameter2 Another parameter description. \* \* @\Doctrine\Orm\Mapper\Entity() \* \* @return string \*/ function test($parameter1, $parameter2) { }

Or even omit the tags section as well (though in the following example is not encouraged as you are missing information on the parameters and return value):

/\*\* \* This is a Summary. \*/ function test($parameter1, $parameter2) { }

A DocBlock may also span a single line as shown in the following example.

/\*\* @var \ArrayObject $array \*/ public $array = null;

Some tags may even feature an "Inline PHPDoc" as shown in the following example.

/\*\* \* @method integer MyMagicMethod(string $argument1) { \* This is the summary for MyMagicMethod. \* \* @param string $argument1 Description for argument 1. \* \* @return integer \* } \*/ class MyMagicClass { ... }

## 6. Inheritance

PHPDoc's also have the ability to inherit information when the succeeding "Structural Element" has a super-element (such as a super-class or a method with the same name in a super-class or implemented in a super-interface).

Every "Structural Element" MUST inherit the following PHPDoc parts by default:

* [Summary](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#51-summary)
* [Description](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#52-description)
* A specific subset of [tags](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#53-tags)
  + [@version](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#825-version)
  + [@author](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#82-author)
  + [@copyright](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#84-copyright)

Each specific "Structural Element" MUST also inherit a specialized subset as defined in the sub-chapters.

The PHPDoc parts MUST NOT be inherited when a replacement is available in the sub-element. The exception to this rule is when the {@inheritdoc} inline tag is present in the Description. When present the parser MUST insert the super-element's Description at the location of the {@inheritdoc} inline tag, while still including the current element's description.

Inheritance takes place from the root of a class hierarchy graph to its leafs. This means that anything inherited in the bottom of the tree MUST 'bubble' up to the top unless overridden.

Note: a special circumstance here would be when the Description must be overridden but the Summary should stay intact. It would be difficult for a reader to distinguish which is overridden.

In this case the writer MUST use the {@inheritdoc} inline tag as Summary and override the Description with the intended text.

Without the {@inheritdoc} inline tag the reader MUST interpret any text as if the Summary would be overridden and Description MAY appear overridden if the block of text contains a Summary ending as defined in the ABNF.

### 6.1. Class Or Interface

In addition to the inherited descriptions and tags as defined in this chapter's root, a class or interface MUST inherit the following tags:

* [@package](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#812-package)

A class or interface SHOULD inherit the following deprecated tags if supplied:

* [@subpackage](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#819-subpackage-deprecated)

The @subpackage MUST NOT be inherited if the @package name of the super-class (or interface) is not the same as the @package of the child class (or interface).

Example:

/\*\* \* @package Framework \* @subpackage Controllers \*/ class Framework\_ActionController { <...> } /\*\* \* @package My \* class My\_ActionController extends Framework\_ActionController { <...> }

In the example above the My\_ActionController MUST NOT inherit the subpackage Controllers.

### 6.2. Function Or Method

In addition to the inherited descriptions and tags as defined in this chapter's root, a function or method in a class or interface MUST inherit the following tags:

* [@param](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#813-param)
* [@return](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#815-return)
* [@throws](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#820-throws)

### 6.3. Constant Or Property

In addition to the inherited descriptions and tags as defined in this chapter's root, a constant or property in a class MUST inherit the following tags:

* [@var](https://github.com/phpDocumentor/fig-standards/blob/master/proposed/phpdoc.md#822-type)

## 7. Describing hashes

The structure of a hash may be described using an "Inline PHPDoc" as part of a @var, @param or @return declaration or using the @struct tag in the Class' DocBlock.

In either case each element of the hash is denoted with a @var declaration in the "Inline PHPDoc". Using this tag it is possible to indicate type, name and purpose of the element.

Please note that the variable name part of the @var tag still needs to be preceded by a dollar sign for readability and parsability of the tag.

Example:

/\*\* \* Initializes this class with the given options. \* \* @param array $options { \* @var boolean $required Whether this element is required \* @var string $label The display name for this element \* } \*/ public function \_\_construct(array $options = array()) { <...> }

### As @struct declaration

In some cases a hash should be documented multiple times in the same class. For these purposes you COULD declare it as a 'virtual' "Structural Element" using the @struct tag in the declaration of a Class or Interface.

It is RECOMMENDED to use native language constructs in these situations, such as a class.

Please see the @struct documentation on how to use this tag.

## 8. Tags

Unless specifically mentioned in the description each tag MAY occur zero or more times in each "DocBlock".

### 8.1. @api

The @api tag is used to declare "Structural Elements" as being suitable for consumption by third parties.

#### Syntax

@api

#### Description

The @api tag represents those "Structural Elements" with a public visibility which are intended to be the public API components for a library or framework. Other "Structural Elements" with a public visibility serve to support the internal structure and are not recommended to be used by the consumer.

The exact meaning of "Structural Elements" tagged with @api MAY differ per project. It is however RECOMMENDED that all tagged "Structural Elements" SHOULD NOT change after publication unless the new version is tagged as breaking Backwards Compatibility.

#### Examples

/\*\* \* This method will not change until a major release. \* \* @api \* \* @return void \*/ function showVersion() { <...> }

### 8.2. @author

The @author tag is used to document the author of any "Structural Element".

#### Syntax

@author [name] [<email address>]

#### Description

The @author tag can be used to indicate who has created a "Structural Element" or has made significant modifications to it. This tag MAY also contain an e-mail address. If an e-mail address is provided it MUST follow the author's name and be contained in chevrons, or angle brackets, and MUST adhere to the syntax defined in RFC 2822.

#### Examples

/\*\* \* @author My Name \* @author My Name <my.name@example.com> \*/

### 8.3. @category [deprecated]

The @category tag is used to organize groups of packages together but is deprecated in favour of occupying the top-level with the @package tag. As such, usage of this tag is NOT RECOMMENDED.

#### Syntax

@category [description]

#### Description

The @category tag was meant in the original de-facto Standard to group several @packages into one category. These categories could then be used to aid in the generation of API documentation.

This was necessary since the @package tag as defined in the original Standard did not contain more then one hierarchy level; since this has changed this tag SHOULD NOT be used.

Please see the documentation for @package for details of its usage.

This tag MUST NOT occur more than once in a "DocBlock".

#### Examples

/\*\* \* File-Level DocBlock \* \* @category MyCategory \* @package MyPackage \*/

### 8.4. @copyright

The @copyright tag is used to document the copyright information of any "Structural element".

#### Syntax

@copyright <description>

#### Description

The @copyright tag defines who holds the copyright over the "Structural Element". The copyright indicated with this tag applies to the "Structural Element" to which it applies and all child elements unless otherwise noted.

The format of the description is governed by the coding standard of each individual project. It is RECOMMENDED to mention the year or years which are covered by this copyright and the organization involved.

#### Examples

/\*\* \* @copyright 1997-2005 The PHP Group \*/

### 8.5. @deprecated

The @deprecated tag is used to indicate which 'Structural elements' are deprecated and are to be removed in a future version.

#### Syntax

@deprecated [<"Semantic Version">][:<"Semantic Version">] [<description>]

#### Description

The @deprecated tag declares that the associated 'Structural elements' will be removed in a future version as it has become obsolete or its usage is otherwise not recommended.

This tag MAY specify up to two version numbers in the sense of a version number range:

The first version number, referred to as the 'starting version', denotes the version in which the associated element has been deprecated.

The second version number, referred to as the 'ending version', denotes the version in which the associated element is scheduled for removal.

If an 'ending version' has been specified, the associated 'Structural elements' MAY no longer exist in the 'ending version' and MAY be removed without further notice in that version or a later version, but MUST exist in all prior versions.

It is RECOMMENDED to specify both a 'starting version' and an 'ending version'. In this case, the two version numbers MUST be separated by a colon (:) without white-space in between.

The 'starting version' MAY be omitted. In this case, the 'ending version' MUST be preceded by a colon.

This tag MAY provide an additional description stating why the associated element is deprecated.

If the associated element is superseded by another it is RECOMMENDED to add a @see tag in the same 'PHPDoc' pointing to the new element.

#### Examples

/\*\* \* @deprecated \* \* @deprecated 1.0.0:2.0.0 \* @see \New\Recommended::method() \* \* @deprecated 1.0.0 \* \* @deprecated :2.0.0 \* \* @deprecated No longer used by internal code and not recommended. \* \* @deprecated 1.0.0 No longer used by internal code and not recommended. \*/

### 8.6. @example

The @example tag is used to link to an external source code file which contains an example of use for the current "Structural element". An inline variant exists with which code from an example file can be shown inline with the Description.

#### Syntax

@example [URI] [<description>]

or inline:

{@example [URI] [:<start>..<end>]}

#### Description

The example tag refers to a file containing example code demonstrating the purpose and use of the current "Structural element". Multiple example tags may be used per "Structural element" in case several scenarios are described.

The URI provided with the example tag is resolved according to the following rules:

1. If a URI is proceeded by a scheme or root folder specifier such as phar://, http://, / or C:\ then it is considered to be an absolute path.
2. If the URI is deemed relative and a location for the example files has been provided then the path relative to the given location is resolved.
3. If the previous path was not readable or the user has not provided a path then the application should try to search for a folder 'examples' in the same folder as the source file featuring the example tag. If found then an attempt to resolve the path by combining the relative path given in the example tag and the found folder should be made.
4. If the application was unable to resolve a path given the previous rules then it should check if a readable folder 'examples' is found in the root folder of the project containing the source file of the "Structural Element".

The root folder of a project is the highest folder common to all files that are being processed by a consuming application.

If a consumer intends to display the contents of the example file then it is RECOMMENDED to use a syntax highlighting solution to improve user experience.

The rules as described above also apply to the inline tags. The inline tag has 2 additional parameters with which to limit which lines of code are shown in the Description. Due to this, consuming applications MUST show the example code in case an inline example tag is used.

The start and end argument may be omitted but the ellipsis should remain in case either is used to give a clear visual indication. The same rules as specified with the [substr](https://php.net/manual/function.substr.php) function of PHP are in effect with regards to the start and end limit.

A consuming application MAY choose to support the limit format as used in the previous standard but it is deprecated per this PSR. The previous syntax was: {@example [URI] [] []} and did not support the same rules as the substr function.

#### Examples

/\*\* \* Counts the number of items. \* {@example http://example.com/foo-inline.https:2..8} \* \* @example http://example.com/foo.phps \* \* @return integer Indicates the number of items. \*/ function count() { <...> }

### 8.7. @global

TODO: The definition of this item should be discussed and whether it may or may not be superceded in part or in whole by the @var tag.

The @global tag is used to denote a global variable or its usage.

#### Syntax

@global ["Type"] [name] @global ["Type"] [description]

#### Description

Since there is no standard way to declare global variables, a @global tag MAY be used in a DocBlock preceding a global variable's definition. To support previous usages of @global, there is an alternate syntax that applies to DocBlocks preceding a function, used to document usage of global variables. In other words, there are two usages of @global: definition and usage.

##### Syntax for the Global's Definition

Only one @global tag MAY be allowed per global variable DocBlock. A global variable DocBlock MUST be followed by the global variable's definition before any other element or DocBlock occurs.

The name MUST be the exact name of the global variable as it is declared in the source.

##### Syntax for the Global's Usage

The function/method @global syntax MAY be used to document usage of global variables in a function, and MUST NOT have a $ starting the third word. The "Type" will be ignored if a match is made between the declared global variable and a variable documented in the project.

#### Examples

(TODO: Examples for this tag should be added)

### 8.8. @internal

The @internal tag is used to denote that the associated "Structural Element" is a structure internal to this application or library. It may also be used inside a description to insert a piece of text that is only applicable for the developers of this software.

#### Syntax

@internal

or inline:

{@internal [description]}}

The inline version of this tag may, contrary to other inline tags, contain text but also other inline tags. To increase readability and ease parsing the tag should be terminated with a double closing brace, instead of a single one.

#### Description

The @internal tag can be used as counterpart of the @api tag, indicating that the associated "Structural Element" is used purely for the internal workings of this piece of software.

When generating documentation from PHPDoc comments it is RECOMMENDED to hide the associated element unless the user has explicitly indicated that internal elements should be included.

An additional use of @internal is to add internal comments or additional description text inline to the Description. This may be done, for example, to withhold certain business-critical or confusing information when generating documentation from the source code of this piece of software.

#### Examples

Mark the count function as being internal to this project:

/\*\* \* @internal \* \* @return integer Indicates the number of items. \*/ function count() { <...> } /\*\* \* Counts the number of Foo. \* \* {@internal Silently adds one extra Foo to compensate for lack of Foo }} \* \* @return integer Indicates the number of items. \*/ function count() { <...> }

### 8.9. @license

The @license tag is used to indicate which license is applicable for the associated 'Structural Elements'.

#### Syntax

@license [<SPDX identifier>|URI] [name]

#### Description

The @license tag provides licensing information to the user, which is applicable to 'Structural Elements' and their child elements.

The first parameter MUST be either a 'SPDX identifier', as defined by the [SPDX Open Source License Registry](https://www.spdx.org/licenses), or a URL to a document containing the full license text.

The second parameter MAY be the official name of the applicable license.

It is RECOMMENDED to only specify an 'SPDX identifier' and to apply @license tags to file-level 'PHPDoc' only, since multiple varying licenses within a single file may cause confusion with regard to which license applies at which time.

In case multiple licenses apply, there MUST be one @license tag per applicable license.

#### Examples

/\*\* \* @license MIT \* \* @license GPL-2.0+ \* \* @license http://www.spdx.org/licenses/MIT MIT License \*/

### 8.10. @link [deprecated]

This tag is deprecated in favor of the@seetag, which since this specification may relate to URIs.

The @link tag indicates a custom relation between the associated "Structural Element" and a website, which is identified by an absolute URI.

#### Syntax

@link [URI] [description]

or inline

@link [URI] [description]

#### Description

The @link tag can be used to define a relation, or link, between the "Structural Element", or part of the description when used inline, to an URI.

The URI MUST be complete and welformed as specified in [RFC 2396](https://tools.ietf.org/html/rfc2396).

The @link tag MAY have a description appended to indicate the type of relation defined by this occurrence.

#### Examples

/\*\* \* @link http://example.com/my/bar Documentation of Foo. \* \* @return integer Indicates the number of items. \*/ function count() { <...> } /\*\* \* This method counts the occurrences of Foo. \* \* When no more Foo ({@link http://example.com/my/bar}) are given this \* function will add one as there must always be one Foo. \* \* @return integer Indicates the number of items. \*/ function count() { <...> }

### 8.11. @method

The @method allows a class to know which 'magic' methods are callable.

#### Syntax

@method [return type] [name]([type] [parameter], [...]) [description]

#### Description

The @method tag is used in situation where a class contains the \_\_call() magic method and defines some definite uses.

An example of this is a child class whose parent has a \_\_call() to have dynamic getters or setters for predefined properties. The child knows which getters and setters need to be present but relies on the parent class to use the\_\_call() method to provide it. In this situation, the child class would have a @method tag for each magic setter or getter method.

The @method tag allows the author to communicate the type of the arguments and return value by including those types in the signature.

When the intended method does not have a return value then the return type MAY be omitted; in which case 'void' is implied.

@method tags MUST NOT be used in a PHPDoc that is not associated with a class or interface.

#### Examples

class Parent { public function \_\_call() { <...> } } /\*\* \* @method string getString() \* @method void setInteger(integer $integer) \* @method setString(integer $integer) \*/ class Child extends Parent { <...> }

### 8.12. @package

The @package tag is used to categorize "Structural Elements" into logical subdivisions.

#### Syntax

@package [level 1]\[level 2]\[etc.]

#### Description

The @package tag can be used as a counterpart or supplement to Namespaces. Namespaces provide a functional subdivision of "Structural Elements" where the @package tag can provide a logical subdivision in which way the elements can be grouped with a different hierarchy.

If, across the board, both logical and functional subdivisions are equal is it NOT RECOMMENDED to use the @package tag, to prevent maintenance overhead.

Each level in the logical hierarchy MUST separated with a backslash (\) to be familiar to Namespaces. A hierarchy MAY be of endless depth but it is RECOMMENDED to keep the depth at less or equal than six levels.

Please note that the @package applies to different "Structural Elements" depending where it is defined.

1. If the @package is defined in the file-level DocBlock then it only applies to the following elements in the applicable file:
   * global functions
   * global constants
   * global variables
   * requires and includes
2. If the @package is defined in a namespace-level or class-level DocBlock then the package applies to that namespace, class or interface and their contained elements. This means that a function which is contained in a namespace with the @package tag assumes that package.

This tag MUST NOT occur more than once in a "DocBlock".

#### Examples

/\*\* \* @package PSR\Documentation\API \*/

### 8.13. @param

The @param tag is used to document a single parameter of a function or method.

#### Syntax

@param ["Type"] [name] [<description>]

#### Description

With the @param tag it is possible to document the type and function of a single parameter of a function or method. When provided it MUST contain a "Type" to indicate what is expected; the description on the other hand is OPTIONAL yet RECOMMENDED. For complex structures such as option arrays it is RECOMMENDED to use an "Inline PHPDoc" to describe the option array.

The @param tag MAY have a multi-line description and does not need explicit delimiting.

It is RECOMMENDED when documenting to use this tag with every function and method. Exceptions to this recommendation are:

This tag MUST NOT occur more than once per parameter in a "PHPDoc" and is limited to "Structural Elements" of type method or function.

#### Examples

/\*\* \* Counts the number of items in the provided array. \* \* @param mixed[] $array Array structure to count the elements of. \* \* @return int Returns the number of elements. \*/ function count(array $items) { <...> }

The following example demonstrates the use of an "Inline PHPDoc" to document an option array with 2 elements: 'required' and 'label'.

/\*\* \* Initializes this class with the given options. \* \* @param array $options { \* @var boolean $required Whether this element is required \* @var string $label The display name for this element \* } \*/ public function \_\_construct(array $options = array()) { <...> }

### 8.14. @property

The @property tag allows a class to know which 'magic' properties are present.

#### Syntax

@property ["Type"] [name] [<description>]

#### Description

The @property tag is used in the situation where a class contains the \_\_get() and \_\_set() magic methods and allows for specific names.

An example of this is a child class whose parent has a \_\_get(). The child knows which properties need to be present but relies on the parent class to use the \_\_get() method to provide it. In this situation, the child class would have a @property tag for each magic property.

@property tags MUST NOT be used in a "PHPDoc" that is not associated with a class or interface.

#### Examples

class Parent { public function \_\_get() { <...> } } /\*\* \* @property string $myProperty \*/ class Child extends Parent { <...> }

### 8.15. @return

The @return tag is used to document the return value of functions or methods.

#### Syntax

@return <"Type"> [description]

#### Description

With the @return tag it is possible to document the return type of a function or method. When provided, it MUST contain a "Type" (See Appendix A) to indicate what is returned; the description on the other hand is OPTIONAL yet RECOMMENDED in case of complicated return structures, such as associative arrays.

The @return tag MAY have a multi-line description and does not need explicit delimiting.

It is RECOMMENDED to use this tag with every function and method. An exception to this recommendation, as defined by the Coding Standard of any individual project, MAY be:

**functions and methods without a**return**value**: the @return tag MAY be omitted here, in which case an interpreter MUST interpret this as if @return void is provided.

This tag MUST NOT occur more than once in a "DocBlock" and is limited to the "DocBlock" of a "Structural Element" of a method or function.

#### Examples

/\*\* \* @return integer Indicates the number of items. \*/ function count() { <...> } /\*\* \* @return string|null The label's text or null if none provided. \*/ function getLabel() { <...> }

### 8.16. @see

The @see tag indicates a reference from the associated "Structural Elements" to a website or other "Structural Elements".

#### Syntax

@see [URI | "FQSEN"] [<description>]

#### Description

The @see tag can be used to define a reference to other "Structural Elements" or to a URI.

When defining a reference to another "Structural Elements" you can refer to a specific element by appending a double colon and providing the name of that element (also called the "FQSEN").

A URI MUST be complete and well-formed as specified in [RFC 2396](https://tools.ietf.org/html/rfc2396).

The @see tag SHOULD have a description to provide additional information regarding the relationship between the element and its target. Additionally, the @see tag MAY have a tag specialization to add further definition to this relationship.

#### Examples

/\*\* \* @see number\_of() :alias: \* @see MyClass::$items For the property whose items are counted. \* @see MyClass::setItems() To set the items for this collection. \* @see http://example.com/my/bar Documentation of Foo. \* \* @return integer Indicates the number of items. \*/ function count() { <...> }

### 8.17. @since

The @since tag is used to denote when an element was introduced or modified, using some description of "versioning" to that element.

#### Syntax

@since [<"Semantic Version">] [<description>]

#### Description

Documents the "version" of the introduction or modification of any element.

It is RECOMMENDED that the version matches a semantic version number (x.x.x) and MAY have a description to provide additional information.

This information can be used to generate a set of API Documentation where the consumer is informed which application version is necessary for a specific element.

The @since tag SHOULD NOT be used to show the current version of an element, the @version tag MAY be used for that purpose.

#### Examples

/\*\* \* This is Foo \* @version MyApp 2.1.7 \* @since 2.0.0 introduced \*/ class Foo { /\*\* \* Make a bar. \* \* @since 2.1.5 bar($arg1 = '', $arg2 = null) \* introduced the optional $arg2 \* @since 2.1.0 bar($arg1 = '') \* introduced the optional $arg1 \* @since 2.0.0 bar() \* introduced new method bar() \*/ public function bar($arg1 = '', $arg2 = null) { <...> } }

### 8.18. @struct

TODO: specify details TODO: determine whether this is a correct approach

### 8.19. @subpackage [deprecated]

The @subpackage tag is used to categorize "Structural Elements" into logical subdivisions.

#### Syntax

@subpackage [name]

#### Description

The @subpackage tag MAY be used as a counterpart or supplement to Namespaces. Namespaces provide a functional subdivision of "Structural Elements" where the @subpackage tag can provide a logical subdivision in which way the elements can be grouped with a different hierarchy.

If, across the board, both logical and functional subdivisions are equal is it NOT RECOMMENDED to use the @subpackage tag, to prevent maintenance overhead.

The @subpackage tag MUST only be used in a specific series of DocBlocks, as is described in the documentation for the @package tag.

This tag MUST accompany a @package tag and MUST NOT occur more than once per DocBlock.

#### Examples

/\*\* \* @package PSR \* @subpackage Documentation\API \*/

### 8.20. @throws

The @throws tag is used to indicate whether "Structural Elements" throw a specific type of exception.

#### Syntax

@throws ["Type"] [<description>]

#### Description

The @throws tag MAY be used to indicate that "Structural Elements" throw a specific type of error.

The type provided with this tag MUST represent an object of the class Exception or any subclass thereof.

This tag is used to present in your documentation which error COULD occur and under which circumstances. It is RECOMMENDED to provide a description that describes the reason an exception is thrown.

It is also RECOMMENDED that this tag occurs for every occurrence of an exception, even if it has the same type. By documenting every occurrence a detailed view is created and the consumer knows for which errors to check.

#### Examples

/\*\* \* Counts the number of items in the provided array. \* \* @param mixed[] $array Array structure to count the elements of. \* \* @throws InvalidArgumentException if the provided argument is not of type \* 'array'. \* \* @return int Returns the number of elements. \*/ function count($items) { <...> }

### 8.21. @todo

The @todo tag is used to indicate whether any development activities should still be executed on associated "Structural Elements".

#### Syntax

@todo [description]

#### Description

The @todo tag is used to indicate that an activity surrounding the associated "Structural Elements" must still occur. Each tag MUST be accompanied by a description that communicates the intent of the original author; this could however be as short as providing an issue number.

#### Examples

/\*\* \* Counts the number of items in the provided array. \* \* @todo add an array parameter to count \* \* @return int Returns the number of elements. \*/ function count() { <...> }

### 8.22. @typedef

Allows the author to define a custom type composed of one or more types that may be augmented with key definitions, properties or methods.

#### Syntax

@typedef ["Type"] [<"QCN">] [<"Inline PHPDoc">]

#### Description

Using the @typedef tag it is possible to define a new pseudo-type or associative array definition for use in PHPDoc blocks.

Let's explain this concept by presenting the following use-cases:

1. You want to document the properties of a class that is dynamically constructed, such as the \stdClass coming from json\_decode.
2. You have a configuration array for which you want to document its keys and associated values.
3. You consume a library with magic methods who does not implement the @method tag but still want to document which methods are on it yourself.
4. You want a pseudo-type called Scalar that represents either a string, float, boolean or integer.
5. You have used class\_alias() to create an alias and want PHPDoc to know which class it is based from so that methods and properties could be inherited.

The first parameter for the @typedef tag is the base "Type", including compound types and collection classes, that is the defining "Type" for the second parameter. The second parameter is used to name your pseudo-type, and MUST be a Qualified Class Name.

The second parameter MAY be omitted, in which case an "Inline PHPDoc" MUST be defined. The information in the "Inline PHPDoc" will augment the existing base class. Using this mechanism it is possible to provide additional information with an existing class, such as methods or properties that could or were not documented in the original.

It is also possible to combine multiple "Types" into a single pseudo-type by using the pipe operator (|), the examples section contains an example of use.

##### Location

A @typedef tag MUST always be placed on a DocBlock that belongs to a File or Class.

When associated with a File the type definition is considered to be global and available throughout your project. It is NOT RECOMMENDED to use it in this fashion without due consideration as you are making your documentation harder to read without generator or IDE.

Type definitions that are associated with a Class MUST only be used inside that class, or its descendants, and are considered to have a visibility similar to protected.

##### Adding methods and properties on objects

It is also possible to add new properties or methods using an "Inline PHPDoc", and the @property and @method tags on any object. In this context an object is any Qualified Class Name (QCN) that does not match one of PHP's primitive types. A notable exception is the 'object' keyword, which may have methods and properties added onto it.

#### Examples

##### Providing an alias for another class

An example may be that the \Storage class is aliased using the class\_alias() function as \Session, and the elements of the \Session class must be documented.

The above can be accomplished with the following tag:

@typedef \Storage \Session

##### Defining additional elements on an aliased class

Here is an example where we add a property and a summary on the new \Session class.

@typedef \Storage \Session { This class represents a session that stores user specific information. @property string $session\_id }

##### Combining multiple types into one

An example of combining multiple types may be a class that is regularly stubbed in unit tests:

@typedef \Mockery\MockInterface|\My\DiContainer DicStub

The above example will construct a pseudo-type DicStub that combines the methods and properties of both the MockInterface and the DiContainer.

##### Defining an associative array as pseudo-type

@typedef array \Configuration { @var string $setting1 @var string $setting2 }

### 8.23. @uses

Indicates whether the current "Structural Element" consumes the "Structural Element", or project file, that is provided as target.

#### Syntax

@uses [file | "FQSEN"] [<description>]

#### Description

The @uses tag describes whether any part of the associated "Structural Element" uses, or consumes, another "Structural Element" or a file that is situated in the current project.

When defining a reference to another "Structural Element" you can refer to a specific element by appending a double colon and providing the name of that element (also called the "FQSEN").

Files that are contained in this project can be referred to by this tag. This can be used, for example, to indicate a relationship between a Controller and a template file (as View).

This tag MUST NOT be used to indicate relations to elements outside of the system, so URLs are not usable. To indicate relations with outside elements the @see tag can be used.

Applications consuming this tag, such as generators, are RECOMMENDED to provide a @used-by tag on the destination element. This can be used to provide a bi-directional experience and allow for static analysis.

#### Examples

/\*\* \* @uses \SimpleXMLElement::\_\_construct() \*/ function initializeXml() { <...> }

/\*\* \* @uses MyView.php \*/ function executeMyView() { <...> }

### 8.24. @var

You may use the @var tag to document the "Type" of the following "Structural Elements":

* Constants, both class and global scope
* Properties
* Variables, both global and local scope

#### Syntax

@var ["Type"] [element\_name] [<description>]

#### Description

The @var tag defines which type of data is represented by a value of a Constant, Property or Variable.

Each Constant or Property definition or Variable where the type is ambiguous or unknown SHOULD be preceded by a DocBlock containing the @var tag. Any other variable MAY be preceded with a DocBlock containing the @var tag.

The @var tag MUST contain the name of the element it documents. An exception to this is when property declarations only refer to a single property. In this case the name of the property MAY be omitted.

This is used when compound statements are used to define a series of Constants or Properties. Such a compound statement can only have one DocBlock while several items are represented.

#### Examples

/\*\* @var int $int This is a counter. \*/ $int = 0; // there should be no docblock here $int++;

Or:

class Foo { /\*\* @var string|null Should contain a description \*/ protected $description = null; public function setDescription($description) { // there should be no docblock here $this->description = $description; } }

Another example is to document the variable in a foreach explicitly; many IDEs use this information to help you with auto-completion:

/\*\* @var \Sqlite3 $sqlite \*/ foreach($connections as $sqlite) { // there should be no docblock here $sqlite->open('/my/database/path'); <...> }

Even compound statements may be documented:

class Foo { protected /\*\* \* @var string Should contain a description \*/ $name, /\*\* \* @var string Should contain a description \*/ $description; }

Or constants:

class Foo { const /\*\* \* @var string Should contain a description \*/ MY\_CONST1 = "1", /\*\* \* @var string Should contain a description \*/ MY\_CONST2 = "2"; }

### 8.25. @version

The @version tag is used to denote some description of "versioning" to an element.

#### Syntax

@version ["Semantic Version"] [<description>]

#### Description

Documents the current "version" of any element.

This information can be used to generate a set of API Documentation where the consumer is informed about elements at a particular version.

It is RECOMMENDED that the version number matches a semantic version number as described in the [Semantic Versioning Standard version 2.0](http://www.semver.org/).

Version vectors from Version Control Systems are also supported, though they MUST follow the form:

name-of-vcs: $vector$

A description MAY be provided, for the purpose of communicating any additional version-specific information.

The @version tag MAY NOT be used to show the last modified or introduction version of an element, the @since tag SHOULD be used for that purpose.

#### Examples

/\*\* \* File for class Foo \* @version 2.1.7 MyApp \* (this string denotes the application's overall version number) \* @version @package\_version@ \* (this PEAR replacement keyword expands upon package installation) \* @version $Id$ \* (this CVS keyword expands to show the CVS file revision number) \*/ /\*\* \* This is Foo \*/ class Foo { <...> }

## Appendix A. Types

### ABNF

A Type has the following [ABNF](https://tools.ietf.org/html/rfc5234) definition:

type-expression = type \*("|" type) type = class-name / keyword / array array = (type / array-expression) "[]" / generic array-expression = "(" type-expression ")" generic = collection-type "<" [type-expression "," \*SP] type-expression ">" collection-type = class-name / "array" class-name = ["\"] label \*("\" label) label = (ALPHA / %x7F-FF) \*(ALPHA / DIGIT / %x7F-FF) keyword = "array" / "bool" / "boolean" / "callable" / "double" / "false" / "float" / "int" / "integer" keyword =/ "mixed" / "null" / "object" / "resource" / "self" / "static" / "string" / "true" / "void" keyword =/ "$this"

### Details

When a "Type" is used the user will expect a value, or set of values, as detailed below.

When the "Type" consists of multiple types then these MUST be separated with the vertical bar sign (|). Any interpreter supporting this specification MUST recognize this and split the "Type" before evaluating.

For example: @return int|null

#### Arrays

The value represented by "Type" can be an array. The type MUST be defined following the format of one of the following options:

1. unspecified, no definition of the contents of the represented array is given. Example: @return array
2. specified containing a single type, the Type definition informs the reader of the type of each array value. Only one type is then expected for each value in a given array.

Example: @return int[]

Please note that mixed is also a single type and with this keyword it is possible to indicate that each array value contains any possible type.

1. specified as containing multiple types, the Type definition informs the reader of the type of each array value. Each value can be of any of the given types. Example: @return (int|string)[]
2. specified using the Generics notation, see the next chapter "Collections" for a description on this notation.

#### Collections

The value represented by "Type" can also be a [Collection](https://en.wikipedia.org/wiki/Collection_(abstract_data_type)), a class that contains a list of keys with values. Collections can be denoted using a format derived from Generics in Java; as such aptly named Generics-style notation.

With Generics-style notation it is REQUIRED to specify a class name, or the array keyword, followed by the type of the values enclosed with angular brackets.

Example: to indicate that this element returns an object of class ArrayObject that only contains a series of strings.

@return \ArrayObject<string>

The type of the values in a Collection MAY be another array and even another Collection,

@return \ArrayObject<\ArrayObject<integer>>

A Collection MAY optionally define the type of the keys of said Collection by adding an additional type definition between the angular brackets before the identifier denoting the values' type. These two should be separated by a comma.

Example: to declare an ArrayObject collection containing a list of strings with integer keys.

@return \ArrayObject<integer, string>

The type of a value, or key, MAY consist of several different types, this can be represented by separating each individual type with a vertical bar sign between the angular brackets.

@return \ArrayObject<string|boolean>

### Valid Class Name

A valid class name seen from the context where this type is mentioned. Thus this may be either a Fully Qualified Class Name (FQCN) or if present in a namespace a local name.

It is RECOMMENDED for applications to expand any local name into a FQCN for easier processing and comparisons.

The element to which this type applies is either an instance of this class or an instance of a class that is a (sub-)child to the given class.

Due to the above nature it is RECOMMENDED for applications that collect and shape this information to show a list of child classes with each representation of the class. This would make it obvious for the user which classes are acceptable as type.

### Keyword

A keyword defining the purpose of this type. Not every element is determined by a class but still worthy of classification to assist the developer in understanding the code covered by the DocBlock.

**Note:**

Most of these keywords are allowed as class names in PHP and as such are hard to distinguish from real classes. As such the keywords MUST be lowercase, as most class names start with an uppercase first character, and you SHOULD NOT use classes with these names in your code.

There are more reasons to not name classes with the names of these keywords but that falls beyond the scope of this specification.

The following keywords are recognized by this PSR:

1. string, the element to which this type applies is a string of binary characters.
2. integer or int, the element to which this type applies is a whole number or integer.
3. boolean or bool, the element to which this type applies only has state TRUE or FALSE.
4. float or double, the element to which this type applies is a continuous, or real, number.
5. object, the element to which this type applies is the instance of an undetermined class.
6. mixed, the element to which this type applies can be of any type as specified here. It is not known on compile time which type will be used.
7. array, the element to which this type applies is an array of values.
8. resource, the element to which this type applies is a resource per the [definition of PHP](https://php.net/manual/language.types.resource.php).
9. void, this type is commonly only used when defining the return type of a method or function.

The basic definition is that the element indicated with this type does not contain a value and the user should not rely on any retrieved value.

**Example 1:**

/\*\* \* @return void \*/ function outputHello() { echo 'Hello world'; }

In the example above no return statement is specified and thus the return value is not determined.

**Example 2:**

/\*\* \* @param boolean $hi when true 'Hello world' is echo-ed. \* \* @return void \*/ function outputHello($quiet) { if ($quiet} { return; } echo 'Hello world'; }

In this example the function contains a return statement without a given value. Because there is no actual value specified, this also qualifies as type void.

1. null, the element to which this type applies is a NULL value or, in technical terms, does not exist.

A big difference compared to void is that this type is used in any situation where the described element may at any given time contain an explicit NULL value.

**Example 1:**

/\*\* \* @return null \*/ function foo() { echo 'Hello world'; return null; }

This type is commonly used in conjunction with another type to indicate that it is possible that nothing is returned.

**Example 2:**

/\*\* \* @param boolean $create\_new When true returns a new stdClass. \* \* @return stdClass|null \*/ function foo($create\_new) { if ($create\_new) { return new stdClass(); } return null; }

1. callable, the element to which this type applies is a pointer to a function call. This may be any type of callable as defined in the PHP manual about [pseudo-types](https://php.net/manual/language.pseudo-types.php) or the section on [callable](https://php.net/manual/language.types.callable.php).
2. false or true, the element to which this type applies will have the value TRUE or FALSE. No other value will be returned from this element.
3. self, the element to which this type applies is of the same class as which the documented element is originally contained.

**Example:**

Method c is contained in class A. The DocBlock states that its return value is of type self. As such method creturns an instance of class A.

This may lead to confusing situations when inheritance is involved.

**Example (previous example situation still applies):**

Class B extends class A and does not redefine method c. As such it is possible to invoke method c from classB.

In this situation ambiguity may arise as self could be interpreted as either class A or B. In these cases selfMUST be interpreted as being an instance of the class where the DocBlock containing the self type is written.

In the examples above self MUST always refer to class A, since it is defined with method c in class A.

Due to the above nature it is RECOMMENDED for applications that collect and shape this information to show a list of child classes with each representation of the class. This would make it obvious for the user which classes are acceptable as type.

1. static, the element to which this type applies is of the same class as which the documented element is contained, or when encountered in a subclass is of type of that subclass instead of the original class.

This keyword behaves the same way as the [keyword for late static binding](https://php.net/manual/language.oop5.late-static-bindings.php) (not the static method, property, nor variable modifier) as defined by PHP.

1. $this, the element to which this type applies is the same exact instance as the current class in the given context. As such this type is a stricter version of static as, in addition, the returned instance must not only be of the same class but also the same instance.

This type is often used as return value for methods implementing the [Fluent Interface](https://en.wikipedia.org/wiki/Fluent_interface) design pattern.