PHPDoc

PSR - 5 y 19

Ñ

Carlos Valencia R.

Contenido

Acerca de

Apreciado amigo/a,

Este documento contiene las especificaciones que definen el estándar ***PHPDoc*** y su etiquetas, recoge las PSR -5 y PSR - 19, que se consideran un borrador, es una traducción al Español, he utilizado el excelente servicio de traductor que ofrece el Sr. Google; mi nivel de Inglés es muy limitado, le pido indulgencia por los errores que pueda encontrar, lo he repasado pero...

Espero que le pueda servir y gracias por su atención.

En Barcelona, Julio del 2021
Carlos Valencia R.

Foto de portada: Antoni Gaudí, La Pedrera - Casa Milà, chimeneas.
Foto procedente de: Piqsels - Fotos de alta resolución libres de derechos.

Estándar PHPDoc - PSR 5

Información general

PHPDoc define un estándar oficial para comentar código php.

El estándar ofrece un método que anima a los programadores a definir y comentar los aspectos del código que normalmente se ignoran.

Permite que los generadores de documentos externos como phpDocumentor puedan crear la documentación API en formato claro y fácil de entender.

Permite que algunos IDEs como Zend Studio, NetBeans, Aptana Studio y PhpStorm interpreten los tipos de variables y otras ambigüedades en el lenguaje de programación.

Puede consultar el original en: https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc.md

El propósito principal de este PSR es proporcionar una definición completa y formal del estándar ***PHPDoc***.

Este PSR se desvía de su predecesor, el estándar ***PHPDoc*** de facto asociado con ***phpDocumentor*** 1.x, para proporcionar soporte para funciones más nuevas en el lenguaje PHP y para abordar algunas de las deficiencias de su predecesor.

Este documento NO DEBE:

Describir un estándar para implementar anotaciones a través de ***PHPDoc***. Aunque ofrece una versatilidad que permite crear un PSR posterior basado en las prácticas actuales.

Describir las mejores prácticas o recomendaciones para estándares de codificación sobre la aplicación del estándar ***PHPDoc***. Este documento se limita a una especificación formal de sintaxis e intención.

Convenciones utilizadas en este documento

Las descritas en la RFC 2119

Al final del documento puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

Definiciones

PHPDoc

Es una sección de documentación que proporciona información sobre aspectos de un "***Elemento estructural***".
Es importante tener en cuenta que ***PHPDoc*** y ***DocBlock*** son dos entidades separadas.

DocBlock

Es la combinación de un ***DocComment***, que es un tipo de comentario, y una entidad ***PHPDoc***.
Es la entidad ***PHPDoc*** la que contiene la sintaxis descrita en esta especificación (como la descripción y las etiquetas).

Elemento estructural

Es una colección de construcciones de programación que PUEDEN estar precedidas por un ***DocBlock***.
La colección contiene las siguientes construcciones:

- require(_once)
- include(_once)
- class
- interface
- trait
- function (incluidos los métodos)
- property
- constant
- variables, tanto de ámbito local como global.

Se RECOMIENDA anteponer un *Elemento estructural* con un *DocBlock* donde esté definido y no con cada uso.

Es una práctica común que el *DocBlock* preceda a un *Elemento estructural*, pero también PUEDE estar separado por un número indeterminado de líneas vacías.

Ejemplo:

```
/** @var int $int Esto es un contador. * /
$int = 0;

// no debería haber ningún docblock aquí
$int++;
```

o

```
/**
 * Esta clase actúa como un ejemplo sobre dónde colocar un DocBlock.
 */
class Foo
{
    /** @var string|null $title contiene un título para Foo * /
    protected $title = null;

    /**
     * Establece un título de una sola línea.
     *
     * @param string $title Un texto para el título.
     *
     * @return void
     */
    public function setTitle($title)
    {
        // no debería haber ningún docblock aquí
        $this->title = $title;
    }
}
```

NO SE RECOMIENDA utilizar definiciones compuestas para *Constantes* o *Propiedades*, ya que el manejo de *DocBlock* en estas situaciones puede conducir a resultados inesperados.

Si se utiliza una declaración compuesta, cada elemento DEBE tener un *DocBlock* anterior.

Ejemplo:

```
  class Foo
  {
    protected
      /**
       * @var string Debe contener una descripción
       */
      $name,
      /**
       * @var string Debe contener una descripción
       */
      $description;
  }
```

Un ejemplo de uso que cae más allá del alcance de esta norma es documentar la variable en un *foreach* explícitamente; varios IDE utilizan esta información para ayudar a su función de autocompletar.

Esta norma no cubre este caso específico, ya que cada declaración se considera una declaración de "*Flujo de control*" en lugar de un *Elemento estructural*.

```
/** @var \Sqlite3 $sqlite */
foreach ($connections as $sqlite) {
  // no debería haber ningún docblock aquí
    $sqlite->open('/my/database/path');
```

```
    <...>
}
```

DocComment es un tipo especial de comentario que:

- DEBE comenzar con la secuencia de caracteres `/**` seguida de un carácter de espacio en blanco.
- DEBE terminar con `*/`
- Tener cero o más líneas en el medio.

En el caso de que un *DocComment* abarque varias líneas, cada línea DEBE comenzar con un asterisco (`*`) que DEBE estar alineado con el primer asterisco de la cláusula inicial.

Ejemplo de una sola línea:

```
/** <...> */
```

Ejemplo con multiples líneas:

```
 /**
  * <...>
  */
```

DocBlock

Es un *DocComment* que contiene una única estructura *PHPDoc* y representa la representación básica en el código fuente.

Tag (etiqueta)

Es una pieza única de metainformación con respecto a un *Elemento estructural* o un componente del mismo.

Type (Tipo)

Es la determinación de qué tipo de datos está asociado con un elemento. Esto se usa comúnmente para determinar los valores exactos de argumentos, constantes, propiedades y más.

Consulte el Apéndice A para obtener información más detallada sobre los *type* (tipos).

Vendor-name (namespace superior)

FQSEN

Es una abreviatura de *Fully Qualified Structural Element Name* (Nombre de elemento estructural totalmente calificado).

FQCN

Es una abreviatura de *Fully Qualified Class Name* (Nombre de clase totalmente cualificado)

Esta notación amplía el *FQCN - Fully Qualified Class Name* (Nombre de clase totalmente cualificado) y agrega una notación para identificar miembros de `class/interface/trait` y volver a aplicar los principios del *FQCN* a interfaces, traits, funciones y constantes globales.

Se pueden utilizar las siguientes notaciones por tipo de *Elemento estructural*:

- 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`

Un *FQSEN Fully Qualified Structural Element Name* (Nombre de elemento estructural totalmente calificado) tiene la siguiente definición ABNF:

```
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 / "_")
```

Principios básicos

- Un *PHPDoc* DEBE estar siempre contenido en un *DocComment*; la combinación de estos dos se llama *DocBlock*.
- Un *DocBlock* DEBE preceder directamente a un *Elemento estructural*.

El formato PHPDoc

El formato *PHPDoc* tiene la siguiente definición ABNF:

```
PHPDoc             = [summary] [description] [tags]

summary            = *CHAR (2*CRLF)

description        = 1*(CHAR / inline-tag) 1*CRLF ; cualquier cantidad de caracteres; con etiquetas en
                         línea en el interior

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 )

tag-description    = 1*(CHAR / CRLF)

tag-signature      = "(" *tag-argument ")"

tag-argument       = *SP 1*CHAR [","] *SP
```

En el un capítulo posterior se incluyen ejemplos de uso.

Summary (Resumen)

Un *summary* DEBE contener un resumen del *Elemento estructural* que define el propósito. Se RECOMIENDA que los *summary* abarquen una sola línea o como máximo dos, pero no más.

Un *summary* DEBE terminar con dos saltos de línea secuenciales, a menos que sea el único contenido en *PHPDoc*.

Si se proporciona una *description*, DEBE ir precedida de un *summary*. De lo contrario, la *description* se considerará el *summary*, hasta que se llegue al final del *summary*.

Dado que un *summary* es comparable al título de un capítulo, es beneficioso utilizar el menor formato posible. Como tal, contrariamente a *description* (consulte el capítulo siguiente), no se hace ninguna recomendación para admitir un lenguaje de marcado. Se deja explícitamente a la aplicación de implementación si quiere admitir esto o no.

Description (Descripción)

description es OPCIONAL pero DEBE incluirse cuando el *Elemento estructural*, al que precede este *DocBlock*, contiene más operaciones, u operaciones más complejas, que las que se pueden describir en el *summary*.

Se RECOMIENDA que cualquier aplicación que analice la *description* admita el lenguaje de marcado Markdown para este campo, de modo que el autor pueda proporcionar formato y una forma clara de representar ejemplos de código.

Los usos comunes de *description* son (entre otros):

- Para proporcionar más detalles sobre lo que hace el método que en *summary*.
- Especificar de qué elementos secundarios se compone un array de entrada o salida, o un objeto.
- Proporcionar un conjunto de casos o escenarios de uso común en los que se puede aplicar el *Elemento estructural*.

Tag (Etiqueta)

Las *tag* proporcionan una forma para que los autores proporcionen metadatos concisos sobre el *Elemento estructural* subsiguiente.

Cada *tag* comienza en una nueva línea, seguida de un signo de arroba (`@`) y un nombre de *tag*, seguido de un espacio en blanco y metadatos (incluida una *description*).

Si se proporcionan metadatos, PUEDEN abarcar varias líneas y PODRÍA seguir un formato estricto y, como tal, proporcionar parámetros, según lo dicte el tipo de *tag*. El tipo de *tag* se puede derivar de su nombre.

Por ejemplo:

```
@param string $argumento Este es un parámetro.
```

La *tag* anterior consta de un nombre (`param`) y metadatos (`string $argumento Este es un parámetro.`) Donde los metadatos se dividen en un *type* (`string`), nombre de variable (`$argumento`) y *description* (` Este es un parámetro. `).

La descripción de una *tag* DEBE admitir Markdown como lenguaje de formato. Debido a la naturaleza de Markdown, es legal comenzar la descripción de la etiqueta en la misma línea o en la siguiente e interpretarla de la misma manera.

Por tanto, estas *tag* son semánticamente idénticas:

```
/**
 * @var string Esta es una descripción.
 * @var string Esta es una
 *    descripción.
 * @var string
 *    Esta es una descripción.
 */
```

Una variación de esto es donde, en lugar de una *description*, se usa una *tag-signature*; en la mayoría de los casos, la *tag* será de hecho una *annotation*". La *tag-signature* puede proporcionar a la *annotation* parámetros relacionados con su funcionamiento.

Si hay una *tag-signature*, NO DEBE haber una *description* presente en la misma *tag*.

Los metadatos proporcionados por las *tag* podrían resultar en un cambio del comportamiento real en tiempo de ejecución del *Elemento estructural* subsiguiente, en cuyo caso el término *annotation* se usa comúnmente en lugar de "*tag*".

Tag Name (Nombre de etiqueta)

Los *Tag name* indican qué tipo de información está representada por esa *tag* o, en el caso de las *annotation* , qué comportamiento debe inyectarse en el *Elemento estructural* siguiente.

En apoyo de las *annotation*, se permite introducir un conjunto de *tag* diseñadas específicamente para una aplicación individual o un subconjunto de aplicaciones (y por lo tanto no están cubiertas por esta especificación).

Estas *tag*, o *annotation* , DEBEN proporcionar un espacio de nombres ya sea prefijando el nombre de la *tag* con un namespace estilo PHP, o con un solo nombre de *vendor-name* seguido de un guión.

Ejemplo de un nombre de *tag* con el prefijo de namespace estilo php (la barra de prefijo es OPCIONAL):

```
@\Doctrine\Orm\Mapping\Entity()
```

Nota: El estándar *PHPDoc* NO hace suposiciones sobre el significado de una *tag* a menos que se especifique en este documento o en adiciones o extensiones posteriores.

Esto significa que PUEDE usar un alias de namespace siempre que se proporcione un elemento de namespace de prefijo.

Por lo tanto, lo siguiente también es legal:

```
@Mapping\Entity()
```

Su propia librería o aplicación puede buscar alias de namespace y hacer un *FQCN* a partir de ellos; esto no tiene ningún impacto en este estándar.

Importante: Las herramientas que utilizan el estándar *PHPDoc* PUEDEN interpretar los namespace que están registrados con esa aplicación y aplicar un comportamiento personalizado.

Ejemplo de un nombre de *tag* con el prefijo del *vendor-name* y un guión:

```
@phpdoc-event transformer.transform.pre
```

Los nombres de *tag* que no tienen el *vendor-name* o un namespace DEBEN describirse en el catálogo de etiquetas PSR y/o en cualquier anexo oficial.

Tag Specialization (Especialización en etiquetas)

Para proporcionar un método mediante el cual proporcionar matices a las *tag* definidas en este estándar, pero sin expandir el conjunto base, se PUEDE proporcionar una especialización después del nombre de la *tag* agregando dos puntos seguidos de una string que proporcione una descripción más matizada de la *tag*.

La lista de especializaciones de *tag* admitidas no se mantiene en el catálogo de etiquetas PSR, ya que puede cambiar con el tiempo.

El PSR puede contener una serie de recomendaciones para nombre de *tag*, pero los proyectos son libres de elegir sus propias especializaciones de *tag*.

Importante: Las herramientas que utilizan el estándar *PHPDoc* PUEDEN interpretar las especializaciones de *tag* que están registradas o comprendidas por esa aplicación y aplican un comportamiento personalizado, pero se espera que implementen el nombre de *tag* anterior como se define en el PSR.

Por ejemplo:

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

La *tag* anterior consta de un nombre ('`see`') y una especialización de *tag* ('`unit-test`') y, por lo tanto, define una relación con la prueba unitaria para el método de procedimiento.

Tag Signature (Firma de etiqueta)

Las *tag-signature* se utilizan comúnmente para *annotation* para proporcionar metadatos adicionales específicos de la *tag* actual.

Los metadatos proporcionados pueden influir en el comportamiento de la *annotation* propietaria y, como tal, influir en el comportamiento del *Elemento estructural* subsiguiente.

El contenido de una *tag-signature* debe ser determinado por el *type* de la *tag* (como se describe en *tag-name*) y queda fuera del alcance de esta especificación. Sin embargo, una *tag-signature* NO DEBE ir seguida de una *description* u otra forma de metadatos.

Ejemplos

Los siguientes ejemplos sirven para ilustrar el uso básico de *DocBlock*; Se recomienda leer la lista de *tag* .

Un ejemplo completo podría verse así:

```
/**
 * Este es un summary.
 *
 * Esta es una description. Puede abarcar varias líneas
 * o contener ejemplos de código usando el marcado Markdown
 *
 * @see Markdown
 *
 * @param int        $parameter1 Una description de parámetro.
 * @param \Exception $e          Otra description de parámetro.
 *
 * @\Doctrine\Orm\Mapper\Entity()
 *
 * @return string
 */
function test($parameter1, $e)
{
    ...
}
```

También se permite omitir la *description*:

```
/**
 * Este es un summary.
 *
 * @see Markdown
 *
 * @param int        $parameter1 Una description de parámetro.
 * @param \Exception $parameter2 Otra description de parámetro.
 *
 * @\Doctrine\Orm\Mapper\Entity()
 *
 * @return string
 */
function test($parameter1, $parameter2)
{
    ...
}
```

O incluso omita la sección de *tag* (aunque no se recomienda, ya que le falta información sobre los parámetros y el valor de retorno):

```
/**
 * Este es un summary.
 */
function test($parameter1, $parameter2)
{
    ...
}
```

Un *DocBlock* también puede abarcar una sola línea:

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

Apéndice A - Types ABNF

Un *type* tiene la siguiente definición ABNF:

```
type-expression = type *("|" type) *("&" type)

type  = class-name / keyword / array

array = (type / array-expression) "[]"

array-expression = "(" type-expression ")"

class-name = ["\"] label *("\" label)

label = (ALPHA / %x7F-FF) *(ALPHA / DIGIT / %x7F-FF)

keyword = "array" / "bool" / "callable" / "false" / "float" / "int" / "iterable" / "mixed" / "null" /
          "object" / "resource" / "self" / "static" / "string" / "true" / "void" / "$this"
```

Detalles

Cuando se utiliza un *type*, el usuario esperará un valor, o un conjunto de valores, como se detalla a continuación.

Cuando el *type* consta de varios *type*, estos DEBEN separarse con el signo de barra vertical (`|`) para la unión o el ampersand (`&`) para la intersección. Cualquier intérprete que admita esta especificación DEBE reconocer esto y dividir el *type* antes de evaluarlo.

Ejemplo de *type* de unión:

```
@return int|null
```

Ejemplo de *type* de intersección:

```
@var \MyClass&\PHPUnit\Framework\MockObject\MockObject $myMockObject
```

Arrays

El valor representado por *type* puede ser un array. El *type* DEBE definirse siguiendo el formato de una de las siguientes opciones:

Sin especificar :

No se da una definición del contenido del array representado.

Ejemplo:

```
@return array
```

Especificado Que contiene un solo type :

La definición de ***type*** informa al lector del ***type*** de cada valor del array. Entonces, solo se espera un ***type*** para cada valor en un array dado.

Ejemplo:

```
@return int[]
```

Tenga en cuenta que ***mixed*** también es un ***type*** único y con esta palabra clave es posible indicar que cada valor del array contiene cualquier ***type*** posible.

Especificado como que contiene varios type :

La definición de ***type*** informa al lector del type de cada valor del array. Cada valor puede ser de cualquiera de los ***type*** dados.

Ejemplo:

```
 @return (int|string)[]
```

Class Name

Un ***class-name*** es válido según el contexto en el que se menciona este ***type***. Por lo tanto, puede ser un nombre de clase totalmente calificado (***FQCN***) o un nombre local si está presente en un namespace.

El elemento al que se aplica este ***type*** es una instancia de esa clase o una instancia de una clase que es un (sub) hijo de la clase dada.

Debido a la naturaleza anterior, se RECOMIENDA que las aplicaciones que recopilan y dan forma a esta información muestren una lista de clases secundarias con cada representación de la clase. Esto haría obvio para el usuario qué clases son aceptables como ***type***.

Keyword (Palabra clave)

Una ***keyword*** define el propósito de este type. No todos los elementos están determinados por una clase, pero aún así son dignos de clasificación para ayudar al desarrollador a comprender el código cubierto por ***DocBlock***.

Nota: La mayoría de estas ***keyword*** están permitidas como nombres de clases en PHP y pueden ser difíciles de distinguir de las clases reales. Como tal, las ***keyword*** DEBEN estar en minúsculas, ya que la mayoría de los nombres de clases comienzan con un primer carácter en mayúscula, y NO DEBE usar clases con estos nombres en su código.

Hay más razones para no nombrar clases con los nombres de estas ***keyword***, pero eso queda fuera del alcance de esta especificación.

Este PSR reconoce las siguientes ***keyword***:

bool - el elemento al que se aplica este ***type*** solo tiene estado TRUE o FALSE.

int - el elemento al que se aplica este ***type*** es un número entero o entero.

float - el elemento al que se aplica este ***type*** es un número continuo o real.

string - el elemento al que se aplica este ***type*** es una string de caracteres binarios.

object - el elemento al que se aplica este ***type*** es la instancia de una clase indeterminada.

array - el elemento al que se aplica este *type* es un array de valores.

iterable - el elemento al que se aplica este *type* es un array u objeto Traversable según la definición de PHP.

resource - el elemento al que se aplica este *type* es un recurso según la definición de PHP.

mixed - el elemento al que se aplica este *type* puede ser de cualquier *type* como se especifica aquí. No se sabe en el momento de la compilación qué *type* se utilizará.

void - este *type* se usa comúnmente solo cuando se define el tipo de retorno de un método o función, lo que indica "*no se devuelve nada*", por lo que el usuario no debe confiar en ningún valor devuelto.

Ejemplo 1:

```
/**
 * @return void
 */
function saludar()
{
    echo '¿Como estan ustedes?';
}
```

En el ejemplo anterior, no se especifica ninguna declaración de retorno y, por lo tanto, no se determina el valor de retorno.

Ejemplo 2:

```
/**
 * @param bool $silencio  si es true no hay mensaje
 *
 * @return void
 */
function saludar($silencio)
{
    if ($silencio) {
        return;
    }
    echo '¿Como estan ustedes?';
}
```

En este ejemplo, la función contiene una declaración de retorno sin un valor dado.

Debido a que no hay un valor real especificado, esto también se califica como *type* `void`.

null - el elemento al que se aplica este *type* es un valor `NULL` o, en términos técnicos, no existe.

Una gran diferencia en comparación con `void` es que este *type* se usa en cualquier situación en la que el elemento descrito puede contener en un momento dado un valor `NULL` explícito.

Ejemplo 1:

```
/**
 * @return null
 */
function foo()
{
    echo '¿Como estan ustedes?';
    return null;
}
```

Este *type* se usa comúnmente junto con otro *type* para indicar que es posible que no se devuelve nada.

Ejemplo 2:

```
/**
 * @param bool $ create_new Cuando es true, devuelve una nueva stdClass.
 *
```

```
 * @return stdClass|null
 */
function foo($create_new)
{
    if ($create_new) {
        return new stdClass();
    }
    return null;
}
```

callable - el elemento al que se aplica este *type* es un puntero a una llamada de función. Esto puede ser cualquier tipo de invocable según la definición de PHP.

false o true - el elemento al que se aplica este *type* tendrá el valor `TRUE` o `FALSE`. No se devolverá ningún otro valor.

self - el elemento al que se aplica este *type* es de la misma clase en la que está contenido originalmente el elemento documentado.

Ejemplo:

> El método c está contenido en la clase A. *DocBlock* establece que su valor de retorno es de *type* self. Como tal, el método c devuelve una instancia de la clase A.
>
> Esto puede llevar a situaciones confusas cuando se trata de herencia.

Ejemplo (la situación de ejemplo anterior todavía se aplica):

> La clase B extiende la clase A y no redefine el método c. Como tal, es posible invocar el método c de la clase B.
>
> En esta situación, puede surgir ambigüedad ya que *self* podría interpretarse como clase A o B. En estos casos, self DEBE interpretarse como una instancia de la clase donde está escrito el *DocBlock* que contiene el *type self*.

En los ejemplos anteriores, *self* siempre DEBE referirse a la clase A, ya que se define con el método c en la clase A.

Debido a la naturaleza anterior, se RECOMIENDA que las aplicaciones que recopilan y dan forma a esta información muestren una lista de clases secundarias con cada representación de la clase. Esto haría obvio para el usuario qué clases son aceptables como *type*.

static - el elemento al que se aplica este *type* es de la misma clase en la que está contenido el elemento documentado o, cuando se encuentra en una subclase, es del *type* de esa subclase en lugar de la clase original.

Esta *keyword* se comporta de la misma manera que la *keyword* para el *static binding* (no el método estático, la propiedad ni el modificador de variable) como lo define PHP.

$this - el elemento al que se aplica este *type* es la misma instancia exacta que la clase actual en el contexto dado. Como tal, este *type* es una versión más estricta de *static*, porque la instancia devuelta no solo debe ser de la misma clase sino también de la misma instancia.

Este *type* se utiliza a menudo como valor de retorno para métodos que implementan el patrón de diseño de Fluent Interface.

Tags (Etiquetas) PHPDoc - PSR 19

Información general

Puede consultar el original en: https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc-tags.md

El objetivo principal de este PSR es proporcionar un catálogo completo de las ***tag*** (etiquetas) en el estándar PHPDoc.

Este documento NO DEBE:

- Describir un catálogo de ***annotation***.
- Describir las mejores prácticas o recomendaciones para estándares de codificación sobre la aplicación del estándar PHPDoc. Este documento se limita a una especificación formal de sintaxis e intención.

Convenciones utilizadas en este documento

Las descritas en la RFC 2119

Al final del documento puede encontrar un extracto del RFC 2119 con las palabras mencionadas y su interpretación.

Definiciones

Consulte la sección Definiciones del PSR 5 - PHPDoc, ya que esas definiciones también se aplican aquí. (***PHPDoc***, ***DocBlock***, ***tag***, ***type***, ***summary***,***annotation*** ...etc.)

Sobre la herencia

Un ***PHPDoc*** que está asociado con un "***Elemento estructural***" que implementa, extiende o anula.

Un "***Elemento estructural***" tiene la capacidad de heredar partes de información del ***PHPDoc*** asociado con el "***Elemento estructural***" que implementa, extiende o reemplaza.

El ***PHPDoc*** para cada ***type*** de "***Elemento estructural***" DEBE heredar las siguientes partes si esa parte está ausente:

- *Summary*
- ***Description*** y un subconjunto específico de ***Tags***:
    - *@author*
    - *@copyright*
    - *@version*

El ***PHPDoc*** para cada ***type*** de "***Elemento estructural***" DEBE también heredar un subconjunto especializado de ***tag*** (etiquetas) dependiendo de qué "***Elemento estructural***" esté asociado.

Si un ***PHPDoc*** no incluye una parte, como ***summary*** o ***description***, que está presente en el ***PHPDoc*** de un superelemento, entonces esa parte siempre se hereda implícitamente.

La siguiente es una lista de todos los elementos cuyos ***DocBlocks*** pueden heredar información del ***DocBlock*** de un superelemento:

- ***DocBlock*** de una clase o interfaz puede heredar información de una clase o interfaz que extiende.

- *DocBlock* de una propiedad puede heredar información de una propiedad con el mismo nombre que se declara en una superclase.
- *DocBlock* de un método puede heredar información de un método con el mismo nombre que se declara en una superclase.
- *DocBlock* de un método puede heredar información de un método con el mismo nombre que se declara en una interfaz implementada en la clase actual o que se implementa en una superclase.

Por ejemplo:

Supongamos que tiene un método `\SubClass::myMethod()` y su clase `\SubClass` extiende la clase `\SuperClass`. Y en la clase `\SuperClass` hay un método con el mismo nombre (por ejemplo, `\SuperClass::myMethod`).

Si se aplica lo anterior, *DocBlock* de `\SubClass::myMethod()` heredará cualquiera de las partes mencionadas anteriormente del *PHPDoc* de `\SuperClass::myMethod`. Entonces, si *@version* no se redefinió, se asume que `\SubClass:: myMethod()` tendrá la misma *@version*.

La herencia tiene lugar desde la raíz de jerarquía de clases hasta sus hojas.

Esto significa que cualquier cosa heredada en la parte inferior del árbol DEBE '*bubble*' (burbujear) hacia arriba a menos que se anule.

Haciendo explícita la herencia usando la tag: @inheritDoc

Debido a que la herencia es implícita, puede suceder que no sea necesario incluir un *PHPDoc* con un "*Elemento estructural*".

Esto puede causar confusión, ya que es ambiguo si el *PHPDoc* se omitió a propósito o si el autor del código se había olvidado de agregar documentación.

Para resolver esta ambigüedad, se puede usr *@inheritDoc*r para indicar que este elemento heredará su información de un superelemento.

Ejemplo:

```
/**
 * Este es un summary.
 */
class SuperClass
{
}

/**
 * @inheritDoc
 */
class SubClass extends SuperClass
{
}
```

En el ejemplo anterior, el *summary* de la subclase se puede considerar igual al del elemento SuperClase, que es, por lo tanto, "Este es un *summary* ".

Usar la tag en una línea {@inheritDoc} para aumentar una description

A veces desea heredar la *description* de un superelemento y agregar su propio texto con él para proporcionar información específica a su "*Elemento estructural*". Esto DEBE hacerse usando la *tag* en una línea `{@inheritDoc}`.

La *tag* en una línea `{@inheritDoc}` indicará que en esa ubicación DEBE inyectarse o inferirse de la description del superelemento.

Ejemplo:

```
/**
 * Este es el summary de este elemento.
 *
 * {@inheritDoc}
 *
 * Además esta description contendrá más información que
 * proporcionará una información más detallada y específica para este
```

```
 * elemento.
 */
```

En el ejemplo anterior, se indica que la ***description*** de este PHPDoc es una combinación de la ***description*** del superelemento, indicado por la ***tag*** en una línea `{@inheritDoc}`, y el texto del cuerpo posterior.

Partes específicas heredadas del elemento

Clase o interfaz

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una clase o interfaz DEBE heredar la siguiente ***tag***:

@package

Función o método

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una clase o interfaz DEBE heredar las siguientes ***tag***:

@param

@return

@throws

Constante o propiedad

Además de las ***description*** y ***tag*** heredadas como se definen en este capítulo, una constante o propiedad en una clase DEBE heredar la siguiente ***tag***:

@var

Tags (Etiquetas)

A menos que se mencione específicamente en la description, cada ***tag*** PUEDE aparecer cero o más veces en cada "***DocBlock***".

@api

@api se usa para resaltar "***Elementos estructurales***" como parte de la API pública primaria de un paquete.

Sintaxis: ***@api***

Descripción: ***@api*** PUEDE aplicarse a los "***Elementos Estructurales***" públicos para resaltarlos en la documentación generada, señalando al consumidor los componentes principales de la API pública de una librería o framework.

Otros "***Elementos Estructurales***" con visibilidad pública PUEDEN figurar de manera menos prominente en la documentación generada.

Consulte también ***@internal***, que PUEDE usarse para ocultar los "***Elementos estructurales***" internos de la documentación generada.

Ejemplos:

```
class UserService
{
    /**
```

```
     * Este método es una API pública.
     *
     * @api
     */
    public function getUser()
    {
        <...>
    }

    /**
     * Este método es "alcance del paquete", no una API pública
     */
    public function callMefromAnotherClass()
    {
        <...>
    }
}
```

@author

@author se utiliza para documentar el autor de cualquier "*Elemento estructural*".

Syntaxis: *@author [name] [<email address>]*

Descripción: *@author* se puede utilizar para indicar quién ha creado un "Elemento estructural" o ha realizado modificaciones importantes.

Esta *tag* PUEDE contener también una dirección de correo electrónico. Si se proporciona una dirección de correo electrónico, DEBE seguir al nombre del autor y estar entre corchetes o corchetes angulares, y DEBE adherirse a la sintaxis definida en RFC 2822.

Ejemplo:

```
/**
 * @author Mi nombre
 * @autor Mi nombre <mi_nombre@ejemplo.com>
 */
```

@copyright

@copyright se utiliza para documentar la información de copyright de cualquier "*Elemento estructural*".

Syntaxis: *@copyright <description>*

Descripción: *@copyright* define quién tiene los derechos de autor sobre el "*Elemento estructural*". Los derechos de autor indicados con esta *tag* se aplican al "*Elemento estructural*" al que se aplica y a todos los elementos secundarios a menos que se indique lo contrario.

El formato de la *description* se rige por el estándar de codificación de cada proyecto individual.

Se RECOMIENDA mencionar el año o años que están cubiertos por este derecho de autor y la organización involucrada.

Ejemplo:

```
/**
 * @copyright 1997-2021 The PHP Group
 */
```

@deprecated

@deprecated se utiliza para indicar qué '*Elementos estructurales*' están en desuso y se eliminarán en una versión futura.

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

Descripción: *@deprecated* declara que los '*Elementos estructurales*' asociados se eliminarán en una versión futura, ya que se han vuelto obsoletos o no se recomienda su uso, a partir de la "*Versión semántica*" si se proporciona.

Esta *tag* PUEDE proporcionar una *description* adicional que indique por qué el elemento asociado está obsoleto.

Si el elemento asociado es reemplazado por otro, se RECOMIENDA agregar una *tag @see* en el mismo '*PHPDoc*' apuntando al nuevo elemento.

Ejemplo:

```
/**
 * @deprecated
 *
 * @deprecated 1.0.0
 *
 * @deprecated Ya no se usa por código interno y no se recomienda.
 *
 * @deprecated 1.0.0 Ya no se usa por código interno y no se recomienda.
 */
```

@internal

@internal se utiliza para indicar que el "*Elemento estructural*" asociado es una estructura interna de esta aplicación o librería.

También se puede usar dentro de una *description* para insertar un fragmento de texto que solo es aplicable para los desarrolladores de este software.

Syntaxis: *@internal [description]*

o en una línea: *{@internal [description]}*

A diferencia de otras *tag* en una línea, la versión en una línea de esta *tag* también puede contener otras *tag* en una línea (consulte el segundo ejemplo a continuación).

Descripción: *@internal* indica que el "*Elemento estructural*" asociado está destinado únicamente para su uso dentro de la aplicación, librería o paquete al que pertenece.

Los autores PUEDEN usar esta *tag* para indicar que un elemento con visibilidad pública debe considerarse exento de la API, por ejemplo:

Los autores de librerías PUEDEN considerar que los cambios importantes en los elementos internos están exentos del control de versiones semántico.

Las herramientas de análisis estático PUEDEN indicar el uso de elementos internos de otra librería/paquete con una advertencia o aviso. Al generar documentación a partir de comentarios *PHPDoc*, se RECOMIENDA ocultar el elemento asociado a menos que el usuario haya indicado explícitamente que se deben incluir elementos internos.

Un uso adicional de *@internal* es agregar comentarios internos o texto descriptivo adicional en la línea *description*. Esto se puede hacer, por ejemplo, para retener cierta información crítica o confusa al generar documentación a partir del código fuente del software.

Ejemplos:

Marca la función `count()` como interna de este proyecto:

```
/**
 * @internal
 *
 * @return int Indica el número de elementos.
 */
function count()
{
```

```
    <...>
}
```

Incluye una nota en la *description* que solo se mostrarán en los documentos para desarrolladores.

```
/**
 * Cuenta el número de Foo.
 *
 * Este método obtiene un recuento de Foo.
 * {@internal Los desarrolladores deben tener en cuenta que silenciosamente
 *            agrega un Foo adicional  (see {@link http://example.com}).}
 *
 * @return int Indica el número de items.
 */
function count()
{
    <...>
}
```

@link

@link indica una relación personalizada entre el "*Elemento estructural*" asociado y un sitio web, que se identifica mediante un URI absoluto.

Syntaxis: *@link [URI] [description]*

o en una línea: *{@link [URI] [description]}*

Descripción: *@link* se puede utilizar para definir una relación, o enlace, entre el "*Elemento estructural*", o parte de la *description* cuando se utiliza en una línea, a un URI.

El URI DEBE estar completo y bien formado como se especifica en la RFC 2396.

@link PUEDE tener una *description* adjunta para indicar el tipo de relación definida por esta ocurrencia.

Ejemplos:

```
/**
 * @link http://ejemplo.com/mi/bar Documentación de Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}

/**
 * Este método cuenta las apariciones de Foo.
 *
 * Cuando no hay ningún Foo ({@link http://ejemplo.com/mi/bar})
 * la función agregará uno, ya que siempre debe haber un Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}
```

@method

@method permite que una clase sepa qué métodos mágicos son invocables.

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

Descripción: *@method* se usa en situaciones en las que una clase contiene el método mágico `__call()` y define algunos usos definidos.

Un ejemplo de esto es una clase hija cuyo padre tiene un `__call()` para tener getters o setters dinámicos para propiedades predefinidas. La hija sabe qué *getters* y *setters* deben estar presentes y confía en la clase padre para que le proporcione el método `__call()`. En esta situación, la clase hija tendría una *@method* para cada método mágico *setter* o *getter*.

@method permite al autor comunicar el *type* de argumentos y el valor de retorno al incluir esos *type* en la firma.

Cuando el método deseado no tiene un valor de retorno, entonces el *type* de retorno PUEDE omitirse; en cuyo caso está implícito el "`void`".
@method SOLO se pueden usar en un *PHPDoc* que esté asociado con una clase o interfaz.

Ejemplo:

```
class Parent
{
    public function __call()
    {
        <...>
    }
}

/**
 * @method setInteger(int $integer)
 * @method string getString()
 * @method void setString(int $integer)
 */
class Child extends Parent
{
    <...>
}
```

@package

@package se utiliza para categorizar "*Elementos estructurales*" en subdivisiones lógicas.

Syntaxis: *@package [level 1]\[level 2]\[etc.]*

Descripción: *@package* se puede utilizar como contraparte o complemento de los namespaces.

Los namespaces proporcionan una subdivisión funcional de "*Elementos estructurales*" donde *@package* puede proporcionar una subdivisión lógica en la que los elementos se pueden agrupar con una jerarquía diferente.

Si, en general, las subdivisiones lógicas y funcionales son iguales, NO SE RECOMIENDA utilizar la *@package*, para evitar gastos de mantenimiento.

Cada nivel de la jerarquía lógica DEBE estar separado con una barra invertida (\) para que los namespaces resulten familiares.

Una jerarquía PUEDE tener una profundidad infinita, pero se RECOMIENDA mantener la profundidad menor o igual a seis niveles.

@package se aplica a ese namespace, clase o interfaz y los elementos que contiene. Significa que una función que está contenida en un namespace con *@package* se asume en ese package.

Esta *tag* NO DEBE aparecer más de una vez en un "*DocBlock*".

Ejemplo:

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

@param

@param se usa para documentar un solo parámetro de una función o método.

Syntaxis: *@param ["Type"] [name] [<description>]*

Descripción: Con *@param* es posible documentar el *type* y función de un solo parámetro de una función o método.

Cuando se proporcione, DEBE contener un *type* para indicar lo que se espera.

El *name* es obligatorio solo cuando se omiten algunas *@param* debido a que toda la información útil ya está visible en la propia firma del código.

La *description* es OPCIONAL pero RECOMENDADA, *@param* PUEDE tener una *description* de varias líneas y no necesita delimitación explícita.

Se RECOMIENDA el uso de esta *tag* con cada función y método.

Esta *tag* NO DEBE aparecer más de una vez por parámetro en un "*PHPDoc*" y está limitada a "*Elementos estructurales*" de tipo método o función.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @param mixed[] $items Estructura del array para contar los elementos.
 *
 * @return int Devuelve el número de elementos.
 */
function count(array $items)
{
    <...>
}
```

@property

@property se utiliza para declarar qué propiedades "*mágicas*" son compatibles.

Syntaxis: *@property[<-read|-write>] ["Type"] [name] [<description>]*

Descripción: *@property* se usa cuando una clase (o trait) implementa los métodos "*mágicos*" `__get()` y/o `__set()` para resolver propiedades no literales en tiempo de ejecución.

Las variantes *@property-read* y *@property-write* PUEDEN usarse para indicar propiedades "*mágicas*" que solo se pueden leer o escribir.

Las *@property* SÓLO se pueden usar en un *PHPDoc* que esté asociado con una clase o trait.

Ejemplo:

En el siguiente ejemplo, una clase Usuario implementa el método mágico `__get()` para implementar la propiedad "*mágica*" de solo lectura `$nombre_completo`:

```
/**
 * @property-read string $nombre_completo
 */
class Usuario
{
    /**
```

```
     * @var string
     */
    public $nombre;

    /**
     * @var string
     */
    public $apellido;

    public function __get($nombre)
    {
        if ($nombre === "nombre_completo") {
            return "{$this->nombre} {$this->apellido}";
        }
    }
}
```

@return

@return se utiliza para documentar el valor de retorno de funciones o métodos.

Syntaxis: *@return <"Type"> [description]*

Descripción: Con *@return* es posible documentar el tipo de retorno de una función o método.

Cuando se proporciona, DEBE contener un *type* para indicar lo que se devuelve; la *description*, por otro lado, es OPCIONAL pero RECOMENDADA en caso de estructuras de retorno complicadas, como arrays asociativos.

@return PUEDE tener una *description* de varias líneas y no necesita delimitación explícita.

Se RECOMIENDA utilizar esta *tag* con cada función y método.

Una excepción a esta recomendación, según la define el Estándar de codificación de cualquier proyecto individual, PUEDE ser en funciones y métodos sin un valor de retorno: *@return* PUEDE omitirse, en cuyo caso un intérprete DEBE interpretar esto como si se proporcionara *@return* `void`.

Esta *tag* NO DEBE aparecer más de una vez en un "*DocBlock*" y está limitada al "*DocBlock*" de un "*Elemento estructural*" de un método o función.

Ejemplos:

```
/**
 * @return int Indica el número de elementos.

 */
function count()
{
    <...>
}

/**
 * @return string|null El texto de la tag o null si no se proporciona ninguno.
 */
function getLabel()
{
    <...>
}
```

@see

@see indica una referencia de los "*Elementos estructurales*" asociados a un sitio web u otros "*Elementos estructurales*".

Sintaxis: *@see [URI | "FQSEN"] [<description>]*

Descripción: *@see* se puede utilizar para definir una referencia a otros "*Elementos estructurales*" o a un URI.

Al definir una referencia a otros "*Elementos Estructurales*", puede hacer referencia a un elemento específico agregando dos puntos y proporcionando el nombre de ese elemento (también llamado "*FQSEN*").

Un URI DEBE estar completo y bien formado como se especifica en RFC 2396.

@see DEBE tener una *description* para proporcionar información adicional sobre la relación entre el elemento y su objetivo. Además, *@see* PUEDE tener una especialización de *tag* para agregar más definición a esta relación.

Ejemplo:

```
/**
 * @see number_of() :alias:
 * @see MyClass::$items           Para la propiedad cuyos elementos se cuentan.
 * @see MyClass::setItems()      Para configurar los elementos de esta colección.
 * @see http://example.com/my/bar Documentación de Foo.
 *
 * @return int Indica el número de elementos.
 */
function count()
{
    <...>
}
```

@since

@since se usa para indicar cuándo se introdujo o modificó un elemento, usando alguna description de "*control de versiones*" para ese elemento.

Syntaxis: *@since [<"Semantic Version">] [<description>]*

Descripción: Documenta la "*versión*" de la introducción o modificación de cualquier elemento.

Se RECOMIENDA que la versión coincida con un número de versión semántica (x.x.x) y PUEDE tener una *description* para proporcionar información adicional.

Esta información se puede utilizar para generar un conjunto de documentación API donde se informa al consumidor qué versión de la aplicación es necesaria para un elemento específico.

@since NO DEBE usarse para mostrar la versión actual de un elemento, *@version* PUEDE usarse para ese propósito.

Ejemplo:

```
/**
 * Esto es Foo
 * @version 2.1.7 MiApp
 * @since 2.0.0 iniciada
 */
class Foo
{
    /**
     * Mover.
     *
     * @since 2.1.5 mover($arg1 = '', $arg2 = null)
     *       se incorpora el $arg2 opcional
     * @since 2.1.0 mover($arg1 = '')
     *        se incorpora el $arg1 opcional
     * @since 2.0.0 mover()
     *        se crea el nuevo método mover()
     */
    public function mover($arg1 = '', $arg2 = null)
    {
        <...>
    }
}
```

@throws

@throws se usa para indicar si los "*Elementos estructurales*" arrojan un *type* específico de *Throwable* (excepción o error).

Syntaxis: *@throws ["Type"] [<description>]*

Descripción: *@throws* PUEDE usarse para indicar que los "elementos estructurales" arrojan un *type* específico de error.

El *type* proporcionado con esta *tag* DEBE representar un objeto que es un subtipo *Throwable*.

Esta *tag* se utiliza para presentar en su documentación qué error PODRÍA ocurrir y bajo qué circunstancias. Se RECOMIENDA proporcionar una *description* que describa el motivo por el que se lanza la excepción.

También se RECOMIENDA que esta *tag* se produzca cada vez que ocurra una excepción, incluso si tiene el mismo *type*. Al documentar cada ocurrencia, se crea una vista detallada y el consumidor sabe qué errores verificar.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @param mixed[] $array  Estructura del array para contar los elementos.
 *
 * @throws InvalidArgumentException Si el argumento proporcionado no es del type 'array'.
 *
 * @return int Devuelve el número de elementos.
 */
function count($items)
{
    <...>
}
```

@todo

@todo se usa para indicar si alguna actividad de desarrollo aún debe ejecutarse en los "*Elementos Estructurales*" asociados.

Syntaxis: *@todo [description]*

Descripción: *@todo* se usa para indicar que aún debe ocurrir una actividad que rodea a los "*Elementos Estructurales*" asociados.

Cada *tag* DEBE ir acompañada de una *description* que comunique la intención del autor original; Sin embargo, esto podría ser tan breve como proporcionar un número.

Ejemplo:

```
/**
 * Cuenta el número de elementos del array proporcionado.
 *
 * @todo agrega un parámetro del array a contar
 *
 * @return int Devuelve el número de elementos.
 */
function count()
{
    <...>
}
```

@uses

@uses indica si el "*Elemento estructural*" actual consume el "*Elemento estructural*", o el archivo de proyecto, que se proporciona como destino.

Syntaxis: *@uses [file | "FQSEN"] [<description>]*

Descripción: *@uses* describe si alguna parte del "*Elemento estructural*" asociado usa, o consume, otro "*Elemento estructural*" o un archivo que se encuentra en el proyecto actual.

Al definir una referencia a otro "*Elemento estructural*", puede hacer referencia a un elemento específico agregando dos puntos dobles y proporcionando el nombre de ese elemento (también llamado "*FQSEN*").

Esta *tag* puede hacer referencia a los archivos incluidos en este proyecto. Esto se puede utilizar, por ejemplo, para indicar una relación entre un *controller* (controlador) y un *view* (archivo de plantilla).

Esta *tag* NO DEBE usarse para indicar relaciones con elementos fuera del sistema, por lo que las URL no se pueden usar. Para indicar relaciones con elementos externos, se puede usar *@see*.

Se RECOMIENDA que las aplicaciones que consumen esta *tag*, como los generadores, proporcionen una *@used-by* en el elemento de destino. Esto se puede utilizar para proporcionar una experiencia bidireccional y permitir el análisis estático.

Ejemplos:

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

/**
 * @uses MiView.php
 */
function executeMiView()
{
    <...>
}
```

@var

Puede utilizar la *@var* para documentar el *type* de los siguientes "*Elementos Estructurales*":

- Constantes, tanto de clase como de ámbito global
- Propiedades
- Variables, tanto de ámbito global como local

Syntaxis: *@var ["Type"] [element_name] [<description>]*

Descripción: *@var* define qué *type* de datos está representado por un valor de Constante, Propiedad o Variable.

Cada constante o definición de propiedad o variable donde el *type* es ambiguo o desconocido DEBE ir precedido de un *DocBlock* que contenga *@var* , cualquier otra variable PUEDE ir precedida de un *DocBlock* que contenga *@var*.

@var DEBE contener el nombre del elemento que documenta. Una excepción a esto es cuando las declaraciones de propiedad solo se refieren a una sola propiedad, en este caso, PUEDE omitirse el nombre de la propiedad.

element_name se utiliza cuando se utilizan sentencias compuestas para definir una serie de constantes o propiedades. Una declaración compuesta de este tipo solo puede tener un *DocBlock* mientras se representan varios elementos.

Ejemplos:

```
/** @var int $int Esto es un contador. * /
$int = 0;

// no debería haber ningún docblock aquí
$int++;
```

o:

```
class Foo
{
  /** @var string|null Debe contener una description * /
  protected $description = null;

  public function setDescription($description)
  {
    // no debería haber ningún docblock aquí
      $this->description = $description;
  }
}
```

Otro ejemplo es documentar la variable en un *foreach* explícitamente; muchos IDE usan esta información para ayudarlo con el autocompletado:

```
/** @var \Sqlite3 $sqlite */
foreach ($connections as $sqlite) {
    // no debería haber ningún docblock aquí
    $sqlite->open('/my/database/path');
    <...>
}
```

Incluso las declaraciones compuestas pueden documentarse:

```
class Foo
{
  protected
      /**
       * @var string Debe contener una description
       */
      $name,
      /**
       * @var string Debe contener una description
       */
      $description;
}
```

o constantes:

```
class Foo
{
  const
      /**
       * @var string Debe contener una description
       */
      MY_CONST1 = "1",
      /**
       * @var string Debe contener una description
       */
      MY_CONST2 = "2";
}
```

@version

@version se usa para denotar alguna *description* de "*control de versiones*" de un elemento.

Syntaxis: *@version ["Semantic Version"] [<description>]*

Descripción: *@version* documenta la versión actual de cualquier elemento.

Esta información se puede utilizar para generar un conjunto de documentación API donde se informa al consumidor sobre los elementos de una versión particular.

Se RECOMIENDA que el número de versión coincida con un número de versión semántica como se describe en el Versionado Semántico 2.0.0

Los vectores de versión de los sistemas de control de versiones también son compatibles, aunque DEBEN seguir la forma:

name-of-vcs: $vector$

PUEDE proporcionarse una ***description*** con el fin de comunicar cualquier información adicional específica de la versión.
@version NO PUEDE usarse para mostrar la última versión modificada o de introducción de un elemento, DEBE usarse ***@since*** para ese propósito.

Ejemplo:

```
/**
 * File for class Foo
 * @version 2.1.7 MyApp
 *           (esta string indica el número de versión general de la aplicación)
 * @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
{
  <...>
}
```

RFC 2119 - (Extracto)

Extracto del documento: https://www.rfc-es.org/rfc/rfc2119-es.txt

El estándar usa varias palabras para indicar los requerimientos de la especificación.

Estas palabras a menudo están en mayúsculas.

Estas palabras deberían ser interpretadas de la forma siguiente:

- DEBE Esta palabra, o los términos "OBLIGATORIO" o "DEBERÁ", significa que la definición es un requerimiento insoslayable de la especificación.
- NO DEBE Esta frase, o la frase "NO DEBERÁ", significa que la definición es una prohibición insoslayable de la especificación.
- DEBERÍA Esta palabra, o el adjetivo "RECOMENDADO", significa que pueden existir razones válidas en determinadas circunstancias para ignorar un elemento determinado, pero se deben comprender y sopesar detenidamente todas las implicaciones antes de tomar una decisión diferente.
- NO DEBERÍA Esta frase, o la frase "NO RECOMENDADO", significa que pueden existir razones válidas en determinadas circunstancias en las que el comportamiento en particular sea útil o incluso aconsejable, pero se deben comprender y sopesar detenidamente todas las implicaciones antes de tomar una decisión diferente.
- PUEDE Esta palabra, o el adjetivo "OPCIONAL", significa que un elemento es realmente opcional. Un proveedor puede elegir incluir el elemento porque un mercado en particular lo necesite o porque el proveedor sienta que realza el producto aunque otro proveedor pueda omitir el mismo elemento. Una implementación que no incluya una opción determinada DEBE estar preparada para interoperar con otra implementación que incluya la opción, aunque quizá con reducida funcionalidad. En el mismo orden de cosas, una implementación que incluya una opción en particular DEBE estar preparada para interoperar con otra implementación que no incluya la opción (excepto, por supuesto, para la característica que aporte la opción).