PHP
Guía de estilo
PSR - 1 y 12

Ñ

Carlos Valencia R.

Contenido

Acerca de

Apreciado amigo/a,

Este documento contiene las especificaciones de Estilo de Codificación para PHP, recoge las PSR - 1 y PSR - 12, que se consideran aceptadas, 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.

PSR 1- Basic Coding Standard (Estándar básico de codificación)

En el documentos se utilizan varias palabras para indicar los requisitos de la especificación. Estas palabras están a menudo en mayúsculas y deben interpretarse como se describe en la RFC 2119.

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

Esta sección comprende lo que deben considerarse los elementos de codificación estándar requeridos para garantizar un alto nivel de interoperabilidad técnica entre el código PHP compartido.

Información general

Puede consultar el original en: https://www.php-fig.org/psr/psr-1

Esta sección comprende lo que deben considerarse los elementos de codificación estándar requeridos para garantizar un alto nivel de interoperabilidad técnica entre el código PHP compartido.

- Los archivos DEBEN usar solo las etiquetas `<?php` y `<?=` .
- Los archivos DEBEN usar solo UTF-8 sin BOM (Byte Order Mask) para el código PHP.
- Un archivo DEBE declarar símbolos (clases, funciones, constantes, etc.) o lógica secundaría (generar informes, cambiar la configuración, etc...) pero NO DEBE hacer ambas cosas.
- Los ***namespace*** y las clases DEBEN seguir el PSR de "***autoloading***" PSR: [PSR-0, PSR-4].
- Los nombres de las clases DEBEN declararse en notación `StudlyCaps` (la primera letra de cada palabra en mayúsculas).
- Las constantes de clase DEBEN declararse en mayúsculas con separadores de subrayado.
- Los nombres de métodos DEBEN declararse en `camelCase` (la primera letra de cada palabra en mayúscula a excepción de la primera palabra)

Archivos

Etiquetas PHP

DEBE usar las etiquetas largas `<?php ?>` o las etiquetas cortas `<?= ?>` .

Codificación de caracteres

DEBE usar solo UTF-8 sin BOM.

Efectos secundarios

Un archivo DEBE declarar símbolos (clases, funciones, constantes, etc.) o lógica secundaría (generar informes, cambiar la configuración, etc...) pero NO DEBE hacer ambas cosas.

La frase "***efectos secundarios***" se refiere a la ejecución de lógica que no está directamente relacionada con la declaración de clases, funciones, constantes, etc.,

Los "***efectos secundarios***" incluyen, entre otros: generar resultados, uso explícito de `require` o `include`, conectarse a servicios externos, modificar la configuración de ini, emitir errores o excepciones, modificar variables globales o estáticas, leer o escribir en un archivo y casos similares.

Sigue un ejemplo de declaraciones y efectos secundarios; es decir, un ejemplo de lo que **se debe evitar**:

```php
<?php

//efecto secundario: cambia la configuración de ini
ini_set('error_reporting', E_ALL);

//efecto secundario: carga un archivo
include "file.php";

//efecto secundario: genera salida
echo "<html>\n";

// declaración
function foo()
{
 //cuerpo de la función
}
```

El siguiente no contiene declaraciones con efectos secundarios; es decir, un ejemplo en la forma propuesta:

```php
<?php

//declaración
function foo()
{
 //contenido
}

//la declaración condicional NO es un efecto secundario
if (! function_exists('bar')) {
    function bar()
    {
     //contenido
    }
}
```

Namespace y nombres de clases

Los ***namespace*** y las clases DEBEN seguir el PSR de "***autoloading***" PSR: [PSR-0, PSR-4].

Esto significa que cada clase está en un único archivo y en un ***namespace*** de al menos un nivel.

Los nombres de las clases DEBEN declararse en notación `StudlyCaps` (la primera letra de cada palabra en mayúsculas).

El código escrito para PHP 5.3 y posteriores DEBE usar ***namespace*** formales.

Por ejemplo:

```php
<?php

// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

El código escrito para versiones 5.2.x y anteriores DEBERÍA usar la convención de ***pseudo-namespacing*** de los prefijos de `Vendor_` en los nombres de clases.

```php
<?php

// PHP 5.2.x and earlier:
class Vendor_Model_Foo
```

```
{
}
```

Constantes de clase, propiedades y métodos

El término "***clase***" se refiere a todas las ***clases***, ***interfaces*** y ***traits***.

Constantes

Las constantes de clase DEBEN declararse en mayúsculas con separadores de subrayado.

Por ejemplo:

```
<?php

namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const FECHA_VERSION = '2021-06-16';
}
```

Propiedades

Se evita intencionalmente cualquier recomendación con respecto a los nombres de propiedades.

DEBE aplicar la convención de nomenclatura que decida `$StudlyCaps`, `$camelCase` o `$under_score` de manera coherente dentro de un alcance razonable. Ese alcance puede ser a ***vendor-level*** (nivel de proveedor), ***package-level*** (nivel de paquete), ***class-level*** (nivel de clase) o ***method-level*** (nivel de método).

Métodos

Los nombres de métodos DEBEN declararse en `camelCase` (la primera letra de cada palabra en mayúscula a excepción de la primera palabra)

PSR 12 - Extended Coding Style (Estilo de codificación extendido)

Esta especificación amplía y reemplaza PSR-2, la guía de estilo de codificación y requiere el cumplimiento de PSR-1, el estándar de codificación básico.

Al igual que PSR-2, la intención de esta especificación es reducir la fricción cognitiva al escanear código de diferentes autores.

Lo hace enumerando un conjunto compartido de reglas y expectativas sobre cómo formatear el código PHP.

Este PSR busca proporcionar una forma establecida en que las herramientas de estilo de codificación puedan implementarse, los proyectos pueden declarar la adherencia y los desarrolladores pueden relacionarse fácilmente entre diferentes proyectos.

Cuando varios autores colaboran en varios proyectos, es útil tener un conjunto de pautas para usar entre todos esos proyectos, por lo tanto, el beneficio de esta guía no está en las reglas en sí, sino en compartir esas reglas.

PSR-2 fue aceptado en 2012 y desde entonces se han realizado varios cambios en PHP que tienen implicaciones para las pautas de estilo de codificación. Si bien PSR-2 es muy completo en cuanto a la funcionalidad PHP que existía en el momento de escribir este artículo, la nueva funcionalidad está muy abierta a la interpretación.

Este PSR, por lo tanto, busca aclarar el contenido del PSR-2 en un contexto más moderno con nueva funcionalidad disponible, y hacer que las erratas del PSR-2 sean vinculantes.

Versiones de lenguajes anteriores

A lo largo de este documento, se PUEDE ignorar cualquier instrucción si no existe en las versiones de PHP compatibles con su proyecto.

Ejemplo

Este ejemplo abarca algunas de las reglas siguientes como una descripción general rápida:

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;

use function Vendor\Package\{functionA, functionB, functionC};

use const Vendor\Package\{ConstantA, ConstantB, ConstantC};

class Foo extends Bar implements FooInterface
{
    public function sampleFunction(int $a, int $b = null): array
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // cuerpo del método
    }
}
```

General

Estándar de codificación básico

El código DEBE seguir todas las reglas descritas en PSR-1.

El término *StudlyCaps* en PSR-1 DEBE interpretarse como *PascalCase*, donde la primera letra de cada palabra está en mayúscula, incluida la primera letra.

Archivos

Todos los archivos PHP DEBEN usar el final de línea Unix `LF` (salto de línea) solamente.

Todos los archivos PHP DEBEN terminar con una línea que no esté en blanco, terminada con un solo `LF`.

La etiqueta de cierre `?>` DEBE omitirse de los archivos que solo contienen PHP.

Líneas

NO DEBE haber un límite estricto en la longitud de la línea.

El límite flexible de la longitud de la línea DEBE ser de 120 caracteres.

Las líneas NO DEBEN tener más de 80 caracteres; las líneas más largas que esa DEBERÍAN dividirse en varias líneas posteriores de no más de 80 caracteres cada una.

NO DEBE haber espacios en blanco al final de las líneas.

PUEDEN agregarse líneas en blanco para mejorar la legibilidad y para indicar bloques de código relacionados, excepto donde esté explícitamente prohibido.

NO DEBE haber más de una declaración por línea.

Sangría

El código DEBE usar una sangría de *4 espacios* para cada nivel de sangría, y NO DEBE usar tabulaciones para sangrar.

Keywords and Types (Palabras clave y tipos)

Todos los *type* y *reserved keywords* (palabras clave reservadas) de PHP [Lista de palabras reservadas] [Listado de otras palabras reservadas] DEBEN estar en minúsculas.

Cualquier nuevo *type* y *keyword* que se agregue a futuras versiones de PHP DEBE estar en minúsculas.

DEBE usarse una forma corta de *keyword* de *type*, es decir, `bool` en lugar de `boolean`, `int` en lugar de `integer`, etc.

Declarar Statements, Namespace, and Import Statements

El encabezado de un archivo PHP puede constar de varios bloques diferentes.

Si está presente, cada uno de los bloques a continuación DEBE estar separado por una sola línea en blanco y NO DEBE contener una línea en blanco.

Cada bloque DEBE estar en el orden que se indica a continuación, aunque los bloques que no son relevantes pueden omitirse.

- Abriendo la etiqueta `<?` Php.

- *Docblock* a nivel de archivo.
- Una o más declare statements.
- La declaración de namespace del archivo.
- Una o más declaraciones `use` de importación de clases.
- Una o más declaraciones `use` de importación de funciones.
- Una o más declaraciones `use` de importación de constantes.
- El resto del código en el archivo.

Cuando un archivo contiene una combinación de HTML y PHP, se puede seguir utilizando cualquiera de las secciones anteriores.

Si es así, DEBEN estar presentes en la parte superior del archivo, incluso si el resto del código consiste en una etiqueta PHP de cierre y luego una mezcla de HTML y PHP.

Cuando la etiqueta de apertura Php `<?` está en la primera línea del archivo, DEBE estar en su propia línea sin otras declaraciones a menos que sea un archivo que contenga marcas fuera de las etiquetas de apertura y cierre de PHP.

Las declaraciones *import* nunca DEBEN comenzar con una barra invertida inicial, ya que siempre deben estar completamente calificadas.

El siguiente ejemplo ilustra una lista completa de todos los bloques:

```
<?php

/**
 * Este archivo contiene un ejemplo de estilos de codificación.
 */

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\SomeNamespace\ClassD as D;
use Vendor\Package\AnotherNamespace\ClassE as E;

use function Vendor\Package\{functionA, functionB, functionC};
use function Another\Vendor\functionD;

use const Vendor\Package\{CONSTANT_A, CONSTANT_B, CONSTANT_C};
use const Another\Vendor\CONSTANT_D;

/**
 * FooBar es un ejemplo de class.
 */
class FooBar
{
    // ... código PHP adicional ...
}
```

NO SE DEBEN utilizar *namespaces* compuestos con una profundidad de más de dos. Por lo tanto, la siguiente es la profundidad máxima de composición permitida:

```
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\ClassA,
    SubnamespaceOne\ClassB,
    SubnamespaceTwo\ClassY,
    ClassZ,
};
```

Y no se permitiría lo siguiente:

```php
<?php

use Vendor\Package\SomeNamespace\{
    SubnamespaceOne\AnotherNamespace\ClassA,
    SubnamespaceOne\ClassB,
    ClassZ,
};
```

Cuando desee declarar un ***type*** estricto en archivos que contienen marcas fuera de las etiquetas de apertura y cierre de PHP, la declaración DEBE estar en la primera línea del archivo e incluir una etiqueta PHP de apertura, la declaración de un ***type*** estricto y la etiqueta de cierre.

Por ejemplo:

```php
<?php declare(strict_types=1) ?>
<html>
<body>
    <?php
        // ... código PHP adicional ...
    ?>
</body>
</html>
```

Los ***declare statements*** NO DEBEN contener espacios y DEBEN ser exactamente `declare(strict_types=1)` (con un terminador de punto y coma opcional).

Los ***declare statements*** en bloque están permitidas y DEBEN tener el formato siguiente. Tenga en cuenta la posición de las llaves (`{}`) y el espaciado:

```php
declare(ticks=1) {
    // algún código
}
```

Clases, propiedades y métodos

El término "***clase***" se refiere a todas las ***clases***, ***interfaces*** y ***traits***.

Cualquier llave de cierre NO DEBE ir seguida de ningún comentario o declaración en la misma línea.

Al crear una instancia de una nueva clase, los paréntesis DEBEN estar siempre presentes incluso cuando no se pasen argumentos al constructor.

```php
new Foo();
```

Extend e Implement

Las palabras clave `extends` e `implements` DEBEN declararse en la misma línea que el nombre de la clase.

La llave de apertura para la clase DEBE ir en su propia línea; la llave de cierre para la clase DEBE ir en la siguiente línea después del cuerpo.

Las llaves de apertura DEBEN estar en su propia línea y NO DEBEN ir precedidas o seguidas de una línea en blanco.

Las llaves de cierre DEBEN estar en su propia línea y NO DEBEN ir precedidas de una línea en blanco.

```php
<?php
```

```php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constantes, propiedades, métodos
}
```

Las listas de implementos y, en el caso de interfaces y extensiones PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez. Al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo una interfaz por línea.

```php
<?php

namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constantes, propiedades, métodos
}
```

Uso de traits

La palabra clave `use` usada dentro de las clases para implementar `trait` DEBE declararse en la siguiente línea después de la llave de apertura.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;
}
```

Cada `trait` individual que se importa a una clase DEBE incluirse uno por línea y cada inclusión DEBE tener su propia declaración de importación de uso.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;
use Vendor\Package\SegundoTrait;
use Vendor\Package\TercerTrait;

class ClassName
{
    use PrimerTrait;
    use SegundoTrait;
    use TercerTrait;
}
```

Cuando la clase no tiene nada después de la declaración de importación `use`, la llave de cierre de la clase (`}`) DEBE estar en la siguiente línea después de la declaración `use`.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;
}
```

De lo contrario, DEBE tener una línea en blanco después de la declaración de importación `use`.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\PrimerTrait;

class ClassName
{
    use PrimerTrait;

    private $property;
}
```

Al usar los operadores `insteadof` y `as`, deben usarse de la siguiente manera, tomando nota de la sangría, el espaciado y las nuevas líneas, mas información en php.net Rasgos (Traits).

```php
<?php

class Talker
{
    use A;
    use B {
        A::smallTalk insteadof B;
    }
    use C {
        B::bigTalk insteadof C;
        C::mediumTalk as FooBar;
    }
}
```

Propiedades y constantes

La visibilidad DEBE declararse en todas las propiedades.

La visibilidad DEBE declararse en todas las constantes si la versión mínima de PHP de su proyecto es PHP 7.1 o posterior.

La *keyword* (palabra clave) `var` NO DEBE usarse para declarar una propiedad.

NO DEBE haber más de una propiedad declarada por declaración.

Los nombres de las propiedades NO DEBEN tener un prefijo con un solo subrayado para indicar visibilidad protegida o privada. Es decir, un prefijo de subrayado no tiene ningún significado explícitamente.

DEBE haber un espacio entre la declaración de tipo y el nombre de la propiedad.

Una declaración de propiedad se parece a lo siguiente:

```php
<?php

namespace Vendor\Package;

class ClassName
{
```

```
    public $foo = null;
    public static int $bar = 0;
}
```

Métodos y funciones

La visibilidad DEBE declararse en todos los métodos.

Los nombres de los métodos NO DEBEN tener un prefijo con un solo subrayado para indicar visibilidad protegida o privada. Es decir, un prefijo de subrayado no tiene ningún significado explícitamente.

Los nombres de métodos y funciones NO DEBEN declararse con un espacio después del nombre del método.

La llave de apertura ({)DEBE ir en su propia línea, y la llave de cierre (}) DEBE ir en la siguiente línea que sigue al cuerpo.

NO DEBE haber un espacio después del paréntesis de apertura ((), y NO DEBE haber un espacio antes del paréntesis de cierre ()).

Una declaración de método tiene el siguiente aspecto, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // cuerpo del método
    }
}
```

Una declaración de función se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

function fooBarBaz($arg1, &$arg2, $arg3 = [])
{
    // cuerpo de la función
}
```

Argumentos de método y función

En la lista de argumentos, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

Los argumentos de método y función con valores predeterminados DEBEN ir al final de la lista de argumentos.

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function foo(int $arg1, &$arg2, $arg3 = [])
    {
        // cuerpo del método
    }
}
```

Las listas de argumentos PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez. Al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento por línea.

Cuando la lista de argumentos se divide en varias líneas, el paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

```
<?php

namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // cuerpo de método
    }
}
```

Cuando tenga una declaración de *type* de retorno presente, DEBE haber un espacio después de los dos puntos seguido de la declaración de tipo.

Los dos puntos y la declaración DEBEN estar en la misma línea que el paréntesis de cierre de la lista de argumentos sin espacios entre los dos caracteres.

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(int $arg1, $arg2): string
    {
        return 'foo';
    }

    public function anotherFunction(
        string $foo,
        string $bar,
        int $baz
    ): string {
        return 'foo';
    }
}
```

En las declaraciones de *type* que aceptan valores `NULL`, NO DEBE haber un espacio entre el signo de interrogación y el tipo.

```
<?php

declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName(?string $arg1, ?int &$arg2): ?string
    {
        return 'foo';
    }
}
```

Cuando se usa el operador de referencia `&` antes de un argumento, NO DEBE haber un espacio después, como en el ejemplo anterior.

NO DEBE haber un espacio entre el operador variable de tres puntos y el nombre del argumento:

```
public function process(string $algorithm, ...$parts)
{
```

```
    // procesando
}
```

Al combinar el operador de referencia y el operador de tres puntos `variadic`, NO DEBE haber ningún espacio entre los dos:

```
public function process(string $algorithm, &...$parts)
{
    // procesando
}
```

abstract, final, y static

Cuando están presentes, las declaraciones `abstract` y `final` DEBEN preceder a la declaración de visibilidad.

Cuando está presente, la declaración `static` DEBE ir después de la declaración de visibilidad.

```
<?php

namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // cuerpo del método
    }
}
```

Llamadas a métodos y funciones

Al realizar una llamada a un método o función, NO DEBE haber un espacio entre el método o el nombre de la función y el paréntesis de apertura, NO DEBE haber un espacio después del paréntesis de apertura y NO DEBE haber un espacio antes del paréntesis de cierre.

En la lista de argumentos, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

```
<?php

bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

Las listas de argumentos PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez, al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento por línea.

Un solo argumento dividido en varias líneas (como podría ser el caso de una función o array anónimo) no constituye una división de la lista de argumentos en sí.

```
<?php

$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

```php
<?php

somefunction($foo, $bar, [
  // ...
], $baz);

$app->get('/hola/{nombre}', function ($nombre) use ($app) {
    return 'Hola ' . $app->escape($nombre);
});
```

Estructuras de control

Las reglas generales de estilo para las estructuras de control son las siguientes:

- DEBE haber un espacio después de la *keyword* (palabra clave) de estructura de control
- NO DEBE haber un espacio después del paréntesis de apertura
- NO DEBE haber un espacio antes del paréntesis de cierre
- DEBE haber un espacio entre el paréntesis de cierre y la llave de apertura
- El cuerpo de la estructura DEBE ser sangrado una vez
- El cuerpo DEBE estar en la siguiente línea después de la llave de apertura.
- La llave de cierre DEBE estar en la siguiente línea después del cuerpo.

El cuerpo de cada estructura DEBE estar encerrado por llaves. Esto estandariza el aspecto de las estructuras y reduce la probabilidad de introducir errores a medida que se agregan nuevas líneas al cuerpo.

if, elseif, else

Una estructura `if` se parece a la siguiente, tenga en cuenta la ubicación de paréntesis, espacios y llaves; y que `else` y `elseif` están en la misma línea que la llave de cierre del cuerpo anterior.

```php
<?php

if ($expr1) {
    // if body
} elseif ($expr2) {
    // cuerpo de elseif
} else {
    // cuerpo de else
}
```

La *keyword* (palabra clave) `elseif` DEBE usarse en lugar de `else if` para que todas las *keyword* de control se vean como palabras individuales.

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```php
<?php

if (
    $expr1
    && $expr2
) {
```

```
    // cuerpo del if
} elseif (
    $expr3
    && $expr4
) {
    // cuerpo del elseif
}
```

switch, case

Una estructura `switch` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

La declaración `case` DEBE tener una sangría una vez desde el `switch` , y la *keyword* `break` (u otra *keywords* de finalización) DEBE tener una sangría al mismo nivel que el cuerpo del `case`.

DEBE haber un comentario como `// no break` cuando la interrupción es intencional en un cuerpo de `case` no vacío.

```
<?php

switch ($expr) {
    case 0:
        echo 'Primer case, con break';
        break;
    case 1:
        echo 'Segundo case, que falla';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Tercer case, volver en lugar del break';
        return;
    default:
        echo 'Case predeterminado';
        break;
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez.

Al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

switch (
    $expr1
    && $expr2
) {
    // estructura del cuerpo
}
```

while, do while

Una declaración `while` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

while ($expr) {
    // estructura del cuerpo
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

while (
    $expr1
    && $expr2
) {
    // estructura del cuerpo
}
```

De manera similar, una instrucción `do while` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

do {
    // estructura del cuerpo
} while ($expr);
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera condición DEBE estar en la siguiente línea.

Los operadores booleanos entre condiciones DEBEN estar siempre al principio o al final de la línea, no una combinación de ambos.

```
<?php

do {
    // estructura del cuerpo
} while (
    $expr1
    && $expr2
);
```

for

Una declaración `for` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

for ($i = 0; $i < 10; $i++) {
    // cuerpo del for
}
```

Las expresiones entre paréntesis PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra al menos una vez, al hacerlo, la primera expresión DEBE estar en la siguiente línea.

El paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

```
<?php

for (
    $i = 0;
    $i < 10;
    $i++
) {
    // cuerpo del for
}
```

foreach

Una declaración `foreach` se parece a la siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

foreach ($iterable as $key => $value) {
    // cuerpo del foreach
}
```

try, catch, finally

Un bloque `try-catch-finally` se parece a lo siguiente, tenga en cuenta la ubicación de los paréntesis, los espacios y las llaves.

```
<?php

try {
    // cuerpo del try
} catch (FirstThrowableType $e) {
    // cuerpo del catch
} catch (OtherThrowableType | AnotherThrowableType $e) {
    // cuerpo del catch
} finally {
    // cuerpo de finally
}
```

Operadores

- Las reglas de estilo para los operadores se agrupan por aridad (el número de operandos que toman).
- Cuando se permite el espacio alrededor de un operador, se PUEDEN utilizar varios espacios para facilitar la lectura.
- Todos los operadores que no se describen aquí quedan sin definir.

Operadores unarios

Los operadores de incremento/decremento NO DEBEN tener ningún espacio entre el operador y el operando.

```
$i++;
++$j;
```

Los operadores de *Type casting* (conversión de tipos) NO DEBEN tener ningún espacio entre paréntesis:

```
$intValue = (int) $input;
```

Operadores binarios

Todos los operadores binarios de aritmética, comparación, asignación, bitwise, lógicos, string y de type DEBEN ir precedidos y seguidos de al menos un espacio:

```
if ($a === $b) {
    $foo = $bar ?? $a ?? $b;
} elseif ($a > $b) {
    $foo = $a + $b * $c;
}
```

Operadores ternarios

El operador condicional, también conocido simplemente como operador ternario, DEBE estar precedido y seguido por al menos un espacio alrededor de ambos `?` y `:` :

```
$variable = $foo ? 'foo' : 'bar';
```

Cuando se omite el operando del medio del operador condicional, el operador DEBE seguir las mismas reglas de estilo que otros operadores de comparación binaria:

```
$variable = $foo ?: 'bar';
```

Cierres

Los cierres DEBEN declararse con un espacio después de la palabra clave `function` y un espacio antes y después de la *keyword* `use`.

La llave de apertura DEBE ir en la misma línea, y la llave de cierre DEBE ir en la siguiente línea que sigue al cuerpo.

NO DEBE haber un espacio después del paréntesis de apertura de la lista de argumentos o de la lista de variables, y NO DEBE haber un espacio antes del paréntesis de cierre de la lista de argumentos o la lista de variables.

En la lista de argumentos y la lista de variables, NO DEBE haber un espacio antes de cada coma, y DEBE haber un espacio después de cada coma.

Los argumentos de cierre con valores predeterminados DEBEN ir al final de la lista de argumentos.

Si un tipo de retorno está presente, DEBE seguir las mismas reglas que con las funciones y métodos normales; si la palabra clave `use` está presente, los dos puntos DEBEN seguir a la lista `use` cerrando paréntesis sin espacios entre los dos caracteres.

Una declaración de cierre tiene el siguiente aspecto, tenga en cuenta la ubicación de los paréntesis, las comas, los espacios y las llaves:

```
<?php

$closureWithArgs = function ($arg1, $arg2) {
    // cuerpo
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
    // cuerpo
};

$closureWithArgsVarsAndReturn = function ($arg1, $arg2) use ($var1, $var2): bool {
    // cuerpo
};
```

Las listas de argumentos y las listas de variables PUEDEN dividirse en varias líneas, donde cada línea subsiguiente se sangra una vez, al hacerlo, el primer elemento de la lista DEBE estar en la siguiente línea y DEBE haber solo un argumento o variable por línea.

Cuando la lista final (ya sea de argumentos o variables) se divide en varias líneas, el paréntesis de cierre y la llave de apertura DEBEN colocarse juntos en su propia línea con un espacio entre ellos.

Los siguientes son ejemplos de cierres con y sin listas de argumentos y listas de variables divididas en varias líneas.

```
<?php

$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
```

```
) {
   // cuerpo
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
   // cuerpo
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
   // cuerpo
};

$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
   // cuerpo
};

$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
   // cuerpo
};
```

Tenga en cuenta que las reglas de formato también se aplican cuando el cierre se usa directamente en una función o llamada a un método como argumento.

```
<?php

$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // cuerpo
    },
    $arg3
);
```

Clases anónimas

Las clases anónimas DEBEN seguir las mismas pautas y principios que los cierres en la sección anterior.

```
<?php

$instance = new class {};
```

La llave de apertura PUEDE estar en la misma línea que la *keyword* `class` siempre que la lista de `implements` interfaces no se ajuste. Si la lista de interfaces se ajusta, la llave DEBE colocarse en la línea que sigue inmediatamente a la última interfaz.

```

```

```php
<?php

// en la misma línea
$instance = new class extends \Foo implements \HandleableInterface {
    // contenido de la clase
};

// en la siguiente línea
$instance = new class extends \Foo implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // Contenido de la clase
};
```

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 decisióndiferente.
- 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).