PSR-12: Extended Coding Style

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.

Overview

This specification extends, expands and replaces PSR-2, the coding style guide and requires adherence to PSR-1, the basic coding standard.

Like PSR-2, the intent of this specification is to reduce cognitive friction when scanning code from different authors. It does so by enumerating a shared set of rules and expectations about how to format PHP code. This PSR seeks to provide a set way that coding style tools can implement, projects can declare adherence to and developers can easily relate to between different projects. When various authors collaborate across multiple projects, it helps to have one set of guidelines to be used among all those projects. Thus, the benefit of this guide is not in the rules themselves but the sharing of those rules.

PSR-2 was accepted in 2012 and since then a number of changes have been made to PHP which has implications for coding style guidelines. Whilst PSR-2 is very comprehensive of PHP functionality that existed at the time of writing, new functionality is very open to interpretation. This PSR, therefore, seeks to clarify the content of PSR-2 in a more modern context with new functionality available, and make the errata to PSR-2 binding.

Previous language versions

Throughout this document, any instructions MAY be ignored if they do not exist in versions of PHP supported by your project.

Example

This example encompasses some of the rules below as a quick overview:

```php
<?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()
    {
        // method body
    }
}
```

2. General

2.1 Basic Coding Standard

Code MUST follow all rules outlined in PSR-1.

The term 'StudlyCaps' in PSR-1 MUST be interpreted as PascalCase where the first letter of each word is capitalized including the very first letter.

2.2 Files

All PHP files MUST use the Unix LF (linefeed) line ending only.

All PHP files MUST end with a non-blank line, terminated with a single LF.

The closing `?>` tag MUST be omitted from files containing only PHP.

2.3 Lines

There MUST NOT be a hard limit on line length.

The soft limit on line length MUST be 120 characters.

Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.

There MUST NOT be trailing whitespace at the end of lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code except where explicitly forbidden.

There MUST NOT be more than one statement per line.

2.4 Indenting

Code MUST use an indent of 4 spaces for each indent level, and MUST NOT use tabs for indenting.

2.5 Keywords and Types

All PHP reserved keywords and types [1][2] MUST be in lower case.

Any new types and keywords added to future PHP versions MUST be in lower case.

Short form of type keywords MUST be used i.e. `bool` instead of `boolean`, `int` instead of `integer` etc.

3. Declare Statements, Namespace, and Import Statements

The header of a PHP file may consist of a number of different blocks. If present, each of the blocks below MUST be separated by a single blank line, and MUST NOT contain a blank line. Each block MUST be in the order listed below, although blocks that are not relevant may be omitted.

- Opening `<?php` tag.
- File-level docblock.
- One or more declare statements.
- The namespace declaration of the file.
- One or more class-based `use` import statements.
- One or more function-based `use` import statements.
- One or more constant-based `use` import statements.
- The remainder of the code in the file.

When a file contains a mix of HTML and PHP, any of the above sections may still be used. If so, they MUST be present at the top of the file, even if the remainder of the code consists of a closing PHP tag and then a mixture of HTML and PHP.

When the opening `<?php` tag is on the first line of the file, it MUST be on its own line with no other statements unless it is a file containing markup outside of PHP opening and closing tags.

Import statements MUST never begin with a leading backslash as they must always be fully qualified.

The following example illustrates a complete list of all blocks:

```php
<?php

/**
 * This file contains an example of coding styles.
 */

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 is an example class.
 */
```

```php
class FooBar
{
    // ... additional PHP code ...
}
```

Compound namespaces with a depth of more than two MUST NOT be used. Therefore the following is the maximum compounding depth allowed:

```php
<?php

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

And the following would not be allowed:

```php
<?php

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

When wishing to declare strict types in files containing markup outside PHP opening and closing tags, the declaration MUST be on the first line of the file and include an opening PHP tag, the strict types declaration and closing tag.

For example:

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

Declare statements MUST contain no spaces and MUST be exactly `declare(strict_types=1)` (with an optional semicolon terminator).

Block declare statements are allowed and MUST be formatted as below. Note position of braces and spacing:

```php
declare(ticks=1) {
    // some code
}
```

4. Classes, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

Any closing brace MUST NOT be followed by any comment or statement on the same line.

When instantiating a new class, parentheses MUST always be present even when there are no arguments passed to the constructor.

```
new Foo();
```

4.1 Extends and Implements

The `extends` and `implements` keywords MUST be declared on the same line as the class name.

The opening brace for the class MUST go on its own line; the closing brace for the class MUST go on the next line after the body.

Opening braces MUST be on their own line and MUST NOT be preceded or followed by a blank line.

Closing braces MUST be on their own line and MUST NOT be preceded by a blank line.

```
<?php

namespace Vendor\Package;

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

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constants, properties, methods
}
```

Lists of `implements` and, in the case of interfaces, `extends` MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

```
<?php

namespace Vendor\Package;

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

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constants, properties, methods
}
```

4.2 Using traits

The `use` keyword used inside the classes to implement traits MUST be declared on the next line after the opening brace.

```
<?php

namespace Vendor\Package;
```

```php
use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

Each individual trait that is imported into a class MUST be included one-per-line and each inclusion MUST have its own `use` import statement.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;
use Vendor\Package\SecondTrait;
use Vendor\Package\ThirdTrait;

class ClassName
{
    use FirstTrait;
    use SecondTrait;
    use ThirdTrait;
}
```

When the class has nothing after the `use` import statement, the class closing brace MUST be on the next line after the `use` import statement.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

Otherwise, it MUST have a blank line after the `use` import statement.

```php
<?php

namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;

    private $property;
}
```

When using the `insteadof` and `as` operators they must be used as follows taking note of indentation, spacing, and new lines.

```php
<?php

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

4.3 Properties and Constants

Visibility MUST be declared on all properties.

Visibility MUST be declared on all constants if your project PHP minimum version supports constant visibilities (PHP 7.1 or later).

The `var` keyword MUST NOT be used to declare a property.

There MUST NOT be more than one property declared per statement.

Property names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

There MUST be a space between type declaration and property name.

A property declaration looks like the following:

```php
<?php

namespace Vendor\Package;

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

4.4 Methods and Functions

Visibility MUST be declared on all methods.

Method names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

Method and function names MUST NOT be declared with space after the method name. The opening brace MUST go on its own line, and the closing brace MUST go on the next line following the body. There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis.

A method declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```php
<?php

namespace Vendor\Package;

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

A function declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```php
<?php

function fooBarBaz($arg1, &$arg2, $arg3 = [])
{
    // function body
}
```

4.5 Method and Function Arguments

In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Method and function arguments with default values MUST go at the end of the argument list.

```php
<?php

namespace Vendor\Package;

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

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

When the argument list is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

```php
<?php

namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
```

```
    ) {
        // method body
    }
}
```

When you have a return type declaration present, there MUST be one space after the colon followed by the type declaration. The colon and declaration MUST be on the same line as the argument list closing parenthesis with no spaces between the two characters.

```php
<?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';
    }
}
```

In nullable type declarations, there MUST NOT be a space between the question mark and the type.

```php
<?php

declare(strict_types=1);

namespace Vendor\Package;

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

When using the reference operator `&` before an argument, there MUST NOT be a space after it, like in the previous example.

There MUST NOT be a space between the variadic three dot operator and the argument name:

```php
public function process(string $algorithm, ...$parts)
{
    // processing
}
```

When combining both the reference operator and the variadic three dot operator, there MUST NOT be any space between the two of them:

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

4.6 `abstract`, `final`, and `static`

When present, the `abstract` and `final` declarations MUST precede the visibility declaration.

When present, the `static` declaration MUST come after the visibility declaration.

```
<?php

namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

4.7 Method and Function Calls

When making a method or function call, there MUST NOT be a space between the method or function name and the opening parenthesis, there MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

```
<?php

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

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line. A single argument being split across multiple lines (as might be the case with an anonymous function or array) does not constitute splitting the argument list itself.

```
<?php

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

```php
<?php

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

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello ' . $app->escape($name);
});
```

5. Control Structures

The general style rules for control structures are as follows:

- There MUST be one space after the control structure keyword
- There MUST NOT be a space after the opening parenthesis
- There MUST NOT be a space before the closing parenthesis
- There MUST be one space between the closing parenthesis and the opening brace
- The structure body MUST be indented once
- The body MUST be on the next line after the opening brace
- The closing brace MUST be on the next line after the body

The body of each structure MUST be enclosed by braces. This standardizes how the structures look and reduces the likelihood of introducing errors as new lines get added to the body.

5.1 `if`, `elseif`, `else`

An `if` structure looks like the following. Note the placement of parentheses, spaces, and braces; and that `else` and `elseif` are on the same line as the closing brace from the earlier body.

```php
<?php

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

The keyword `elseif` SHOULD be used instead of `else if` so that all control keywords look like single words.

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

```php
<?php

if (
    $expr1
    && $expr2
```

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

5.2 `switch`, `case`

A `switch` structure looks like the following. Note the placement of parentheses, spaces, and braces. The `case` statement MUST be indented once from `switch`, and the `break` keyword (or other terminating keywords) MUST be indented at the same level as the `case` body. There MUST be a comment such as `// no break` when fall-through is intentional in a non-empty `case` body.

```php
<?php

switch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

```php
<?php

switch (
    $expr1
    && $expr2
) {
    // structure body
}
```

5.3 `while`, `do while`

A `while` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php

while ($expr) {
```

```
    // structure body
}
```

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

```
<?php

while (
    $expr1
    && $expr2
) {
    // structure body
}
```

Similarly, a `do while` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

do {
    // structure body;
} while ($expr);
```

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first condition MUST be on the next line. Boolean operators between conditions MUST always be at the beginning or at the end of the line, not a mix of both.

```
<?php

do {
    // structure body;
} while (
    $expr1
    && $expr2
);
```

5.4 `for`

A `for` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

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

Expressions in parentheses MAY be split across multiple lines, where each subsequent line is indented at least once. When doing so, the first expression MUST be on the next line. The closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

```
<?php

for (
```

```
    $i = 0;
    $i < 10;
    $i++
) {
    // for body
}
```

5.5 `foreach`

A `foreach` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

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

5.6 `try`, `catch`, `finally`

A `try-catch-finally` block looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

try {
    // try body
} catch (FirstThrowableType $e) {
    // catch body
} catch (OtherThrowableType | AnotherThrowableType $e) {
    // catch body
} finally {
    // finally body
}
```

6. Operators

Style rules for operators are grouped by arity (the number of operands they take).

When space is permitted around an operator, multiple spaces MAY be used for readability purposes.

All operators not described here are left undefined.

6.1. Unary operators

The increment/decrement operators MUST NOT have any space between the operator and operand.

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

Type casting operators MUST NOT have any space within the parentheses:

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

6.2. Binary operators

All binary arithmetic, comparison, assignment, bitwise, logical, string, and type operators MUST be preceded and followed by at least one space:

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

6.3. Ternary operators

The conditional operator, also known simply as the ternary operator, MUST be preceded and followed by at least one space around both the `?` and `:` characters:

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

When the middle operand of the conditional operator is omitted, the operator MUST follow the same style rules as other binary comparison operators:

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

7. Closures

Closures MUST be declared with a space after the `function` keyword, and a space before and after the `use` keyword.

The opening brace MUST go on the same line, and the closing brace MUST go on the next line following the body.

There MUST NOT be a space after the opening parenthesis of the argument list or variable list, and there MUST NOT be a space before the closing parenthesis of the argument list or variable list.

In the argument list and variable list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Closure arguments with default values MUST go at the end of the argument list.

If a return type is present, it MUST follow the same rules as with normal functions and methods; if the `use` keyword is present, the colon MUST follow the `use` list closing parentheses with no spaces between the two characters.

A closure declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php

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

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

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

Argument lists and variable lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument or variable per line.

When the ending list (whether of arguments or variables) is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

The following are examples of closures with and without argument lists and variable lists split across multiple lines.

```php
<?php

$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
   // body
};

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

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

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

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

Note that the formatting rules also apply when the closure is used directly in a function or method call as an argument.

```php
<?php

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

8. Anonymous Classes

Anonymous Classes MUST follow the same guidelines and principles as closures in the above section.

```php
<?php

$instance = new class {};
```

The opening brace MAY be on the same line as the `class` keyword so long as the list of `implements` interfaces does not wrap. If the list of interfaces wraps, the brace MUST be placed on the line immediately following the last interface.

```php
<?php

// Brace on the same line
$instance = new class extends \Foo implements \HandleableInterface {
    // Class content
};

// Brace on the next line
$instance = new class extends \Foo implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // Class content
};
```