https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md

Basic Coding Standard

This section of the standard comprises what should be considered the standard coding elements that are required to ensure a high level of technical interoperability between shared PHP code.

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.

1. Overview

- Files MUST use only `<?php` and `<?=` tags.
- Files MUST use only UTF-8 without BOM for PHP code.
- Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side-effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow an "autoloading" PSR: [PSR-0, PSR-4].
- Class names MUST be declared in `StudlyCaps`.
- Class constants MUST be declared in all upper case with underscore separators.
- Method names MUST be declared in `camelCase`.

2. Files

2.1. PHP Tags

PHP code MUST use the long `<?php ?>` tags or the short-echo `<?= ?>` tags; it MUST NOT use the other tag variations.

2.2. Character Encoding

PHP code MUST use only UTF-8 without BOM.

2.3. Side Effects

A file SHOULD declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or it SHOULD execute logic with side effects, but SHOULD NOT do both.

The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., merely from including the file.

"Side effects" include but are not limited to: generating output, explicit use of `require` or `include`, connecting to external services, modifying ini settings, emitting errors or exceptions, modifying global or static variables, reading from or writing to a file, and so on.
The following is an example of a file with both declarations and side effects; i.e, an example of what to avoid:

```php
<?php
// side effect: change ini settings
ini_set('error_reporting', E_ALL);

// side effect: loads a file
include "file.php";

// side effect: generates output
echo "<html>\n";

// declaration
function foo()
{
    // function body
}
```

The following example is of a file that contains declarations without side effects; i.e., an example of what to emulate:

```php
<?php
// declaration
function foo()
{
    // function body
}

// conditional declaration is *not* a side effect
```

```php
if (! function_exists('bar')) {
    function bar()
    {
        // function body
    }
}
```

3. Namespace and Class Names

Namespaces and classes MUST follow an "autoloading" PSR: [PSR-0, PSR-4].

This means each class is in a file by itself, and is in a namespace of at least one level: a top-level vendor name.

Class names MUST be declared in `StudlyCaps`.
Code written for PHP 5.3 and after MUST use formal namespaces.

For example:

```php
<?php
// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

Code written for 5.2.x and before SHOULD use the pseudo-namespacing convention of `Vendor_` prefixes on class names.

```php
<?php
// PHP 5.2.x and earlier:
class Vendor_Model_Foo
{
}
```

4. Class Constants, Properties, and Methods

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

4.1. Constants

Class constants MUST be declared in all upper case with underscore separators. For example:

```php
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

4.2. Properties

This guide intentionally avoids any recommendation regarding the use of `$StudlyCaps`, `$camelCase`, or `$under_score`property names.
Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.

4.3. Methods

Method names MUST be declared in `camelCase()`.

Coding Style Guide

This guide extends and expands on PSR-1, the basic coding standard.

The intent of this guide 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.

The style rules herein are derived from commonalities among the various member 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 in the sharing of those rules.

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.

1. Overview

- Code MUST follow a "coding style guide" PSR [PSR-1].

•Code MUST use 4 spaces for indenting, not tabs.

•There MUST NOT be a hard limit on line length; the soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.

•There MUST be one blank line after the `namespace` declaration, and there MUST be one blank line after the block of `use`declarations.

•Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.

•Opening braces for methods MUST go on the next line, and closing braces MUST go on the next line after the body.

•Visibility MUST be declared on all properties and methods; `abstract` and `final` MUST be declared before the visibility;`static` MUST be declared after the visibility.

•Control structure keywords MUST have one space after them; method and function calls MUST NOT.

•Opening braces for control structures MUST go on the same line, and closing braces MUST go on the next line after the body.

•Opening parentheses for control structures MUST NOT have a space after them, and closing parentheses for control structures MUST NOT have a space before.

1.1. Example

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

```
<?php
namespace Vendor\Package;

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

class Foo extends Bar implements FooInterface
{
```

```
    public function sampleFunction($a, $b = null)
    {
        if ($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.

2.2 Files

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

All PHP files MUST end with a single blank line.

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; automated style checkers MUST warn but MUST NOT error at the soft limit.

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 non-blank lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code.

There MUST NOT be more than one statement per line.

2.4. Indenting

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

N.b.: Using only spaces, and not mixing spaces with tabs, helps to avoid problems with diffs, patches, history, and annotations. The use of spaces also makes it easy to insert fine-grained sub-indentation for inter-line alignment.

2.5. Keywords and True/False/Null

PHP keywords MUST be in lower case.

The PHP constants `true`, `false`, and `null` MUST be in lower case.

3. Namespace and Use Declarations

When present, there MUST be one blank line after the `namespace` declaration.
When present, all `use` declarations MUST go after the `namespace` declaration.
There MUST be one `use` keyword per declaration.
There MUST be one blank line after the `use` block.
For example:

```
<?php
namespace Vendor\Package;

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

// ... additional PHP code ...
```

4. Classes, Properties, and Methods

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

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.

```
<?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` 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. Properties

Visibility MUST be declared on all properties.

The `var` keyword MUST NOT be used to declare a property.
There MUST NOT be more than one property declared per statement.

Property names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

A property declaration looks like the following.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public $foo = null;
}
```

4.3. Methods

Visibility MUST be declared on all methods.

Method names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

Method names MUST NOT be declared with a 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
namespace Vendor\Package;

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

4.4. Method Arguments

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

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

```
<?php
```

```php
namespace Vendor\Package;

class ClassName
{
    public function foo($arg1, &$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
    }
}
```

4.5. `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
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();
```

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

4.6. 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.

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

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 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
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.

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 keyword) 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
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;
}
```

5.3. `while`, `do while`

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

```
<?php
while ($expr) {
    // 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);
```

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
}
```

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`

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

```
<?php
try {
```

```
    // try body
} catch (FirstExceptionType $e) {
    // catch body
} catch (OtherExceptionType $e) {
    // catch body
}
```

6. 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.

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
};
```

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

7. Conclusion

There are many elements of style and practice intentionally omitted by this guide. These include but are not limited to:

- Declaration of global variables and global constants
- Declaration of functions
- Operators and assignment
- Inter-line alignment
- Comments and documentation blocks
- Class name prefixes and suffixes
- Best practices

Future recommendations MAY revise and extend this guide to address those or other elements of style and practice.

Logger Interface

This document describes a common interface for logging libraries.

The main goal is to allow libraries to receive a `Psr\Log\LoggerInterface` object and write logs to it in a simple and universal way. Frameworks and CMSs that have custom needs MAY extend the interface for their own purpose, but SHOULD remain compatible with this document. This ensures that the third-party libraries an application uses can write to the centralized application logs.
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.

The word `implementor` in this document is to be interpreted as someone implementing the `LoggerInterface` in a log-related library or framework. Users of loggers are referred to as `user`.

1. Specification

1.1 Basics

- The `LoggerInterface` exposes eight methods to write logs to the eight RFC 5424 levels (debug, info, notice, warning, error, critical, alert, emergency).
- A ninth method, `log`, accepts a log level as first argument. Calling this method with one of the log level constants MUST have the same result as calling the level-specific method. Calling this method with a level not defined by this specification MUST throw a `Psr\Log\InvalidArgumentException` if the implementation does not know about the level. Users SHOULD NOT use a custom level without knowing for sure the current implementation supports it.

1.2 Message

- Every method accepts a string as the message, or an object with a `__toString()` method. Implementors MAY have special handling for the passed objects. If that is not the case, implementors MUST cast it to a string.

- The message MAY contain placeholders which implementors MAY replace with values from the context array.

Placeholder names MUST correspond to keys in the context array.

Placeholder names MUST be delimited with a single opening brace `{` and a single closing brace `}`. There MUST NOT be any whitespace between the delimiters and the placeholder name.
Placeholder names SHOULD be composed only of the characters `A-Z`, `a-z`, `0-9`, underscore `_`, and period `.`. The use of other characters is reserved for future modifications of the placeholders specification.

Implementors MAY use placeholders to implement various escaping strategies and translate logs for display. Users SHOULD NOT pre-escape placeholder values since they can not know in which context the data will be displayed.

The following is an example implementation of placeholder interpolation provided for reference purposes only:

```php
/**
* Interpolates context values into the message placeholders.
*/
function interpolate($message, array $context = array())
{
    // build a replacement array with braces around the context keys
    $replace = array();
    foreach ($context as $key => $val) {
        $replace['{' . $key . '}'] = $val;
    }

    // interpolate replacement values into the message and return
    return strtr($message, $replace);
}

// a message with brace-delimited placeholder names
$message = "User {username} created";

// a context array of placeholder names => replacement values
$context = array('username' => 'bolivar');

// echoes "User bolivar created"
echo interpolate($message, $context);
```

1.3 Context

•Every method accepts an array as context data. This is meant to hold any extraneous information that does not fit well in a string. The array can contain anything. Implementors MUST ensure they treat context data with as much lenience as possible. A given value in the context MUST NOT throw an exception nor raise any php error, warning or notice.

•If an `Exception` object is passed in the context data, it MUST be in the `'exception'` key. Logging exceptions is a common pattern and this allows implementors to extract a stack trace from the exception when the log backend supports it. Implementors MUST still verify that the `'exception'` key is actually an `Exception` before using it as such, as it MAY contain anything.

1.4 Helper classes and interfaces

•The `Psr\Log\AbstractLogger` class lets you implement the `LoggerInterface` very easily by extending it and implementing the generic `log` method. The other eight methods are forwarding the message and context to it.
•Similarly, using the `Psr\Log\LoggerTrait` only requires you to implement the generic `log` method. Note that since traits can not implement interfaces, in this case you still have to implement `LoggerInterface`.
•The `Psr\Log\NullLogger` is provided together with the interface. It MAY be used by users of the interface to provide a fall-back "black hole" implementation if no logger is given to them. However conditional logging may be a better approach if context data creation is expensive.
•The `Psr\Log\LoggerAwareInterface` only contains a `setLogger(LoggerInterface $logger)` method and can be used by frameworks to auto-wire arbitrary instances with a logger.
•The `Psr\Log\LoggerAwareTrait` trait can be used to implement the equivalent interface easily in any class. It gives you access to `$this->logger`.
•The `Psr\Log\LogLevel` class holds constants for the eight log levels.

2. Package

The interfaces and classes described as well as relevant exception classes and a test suite to verify your implementation is provided as part of the psr/log package.

3. `Psr\Log\LoggerInterface`

```
<?php
```

```php
namespace Psr\Log;

/**
 * Describes a logger instance
 *
 * The message MUST be a string or object implementing __toString().
 *
 * The message MAY contain placeholders in the form: {foo} where foo
 * will be replaced by the context data in key "foo".
 *
 * The context array can contain arbitrary data, the only assumption that
 * can be made by implementors is that if an Exception instance is given
 * to produce a stack trace, it MUST be in a key named "exception".
 *
 * See https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-3-logger-interface.md
 * for the full interface specification.
 */
interface LoggerInterface
{
    /**
     * System is unusable.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function emergency($message, array $context = array());

    /**
     * Action must be taken immediately.
     *
     * Example: Entire website down, database unavailable, etc. This should
     * trigger the SMS alerts and wake you up.
     *
     * @param string $message
     * @param array $context
```

```
     * @return null
     */
    public function alert($message, array $context = array());

    /**
     * Critical conditions.
     *
     * Example: Application component unavailable, unexpected exception.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function critical($message, array $context = array());

    /**
     * Runtime errors that do not require immediate action but should typically
     * be logged and monitored.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function error($message, array $context = array());

    /**
     * Exceptional occurrences that are not errors.
     *
     * Example: Use of deprecated APIs, poor use of an API, undesirable things
     * that are not necessarily wrong.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function warning($message, array $context = array());

    /**
```

```php
     * Normal but significant events.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function notice($message, array $context = array());

    /**
     * Interesting events.
     *
     * Example: User logs in, SQL logs.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function info($message, array $context = array());

    /**
     * Detailed debug information.
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function debug($message, array $context = array());

    /**
     * Logs with an arbitrary level.
     *
     * @param mixed $level
     * @param string $message
     * @param array $context
     * @return null
     */
    public function log($level, $message, array $context = array());
}
```

4. Psr\Log\LoggerAwareInterface

```php
<?php

namespace Psr\Log;

/**
 * Describes a logger-aware instance
 */
interface LoggerAwareInterface
{
    /**
     * Sets a logger instance on the object
     *
     * @param LoggerInterface $logger
     * @return null
     */
    public function setLogger(LoggerInterface $logger);
}
```

5. Psr\Log\LogLevel

```php
<?php

namespace Psr\Log;

/**
 * Describes log levels
 */
class LogLevel
{
    const EMERGENCY = 'emergency';
    const ALERT     = 'alert';
    const CRITICAL  = 'critical';
    const ERROR     = 'error';
    const WARNING   = 'warning';
    const NOTICE    = 'notice';
    const INFO      = 'info';
    const DEBUG     = 'debug';
}
```