PSR stands for PHP Standards Recommendation, which are a set of guidelines and best practices for writing clean, consistent, and maintainable PHP code. There are several PSRs that cover different aspects of PHP development, such as coding style, autoloading, interfaces, and more. Some of the most commonly referenced PSRs include:

|  |  |  |
| --- | --- | --- |
| PSR-0 | Autoloading Standard | It describes the mandatory requirements that must be adhered to for autoloader interoperability. |

* A fully-qualified namespace and class must have the following structure \<Vendor Name>\(<Namespace>\)\*<Class Name>
* Each namespace must have a top-level namespace ("Vendor Name").
* Each namespace can have as many sub-namespaces as it wishes.
* Each namespace separator is converted to a DIRECTORY\_SEPARATOR when loading from the file system.
* Each \_ character in the CLASS NAME is converted to a DIRECTORY\_SEPARATOR. The \_ character has no special meaning in the namespace.
* The fully-qualified namespace and class are suffixed with .php when loading from the file system.
* Alphabetic characters in vendor names, namespaces, and class names may be of any combination of lower case and upper case.

Examples

* \Doctrine\Common\IsolatedClassLoader => /path/to/project/lib/vendor/Doctrine/Common/IsolatedClassLoader.php
* \Symfony\Core\Request => /path/to/project/lib/vendor/Symfony/Core/Request.php
* \Zend\Acl => /path/to/project/lib/vendor/Zend/Acl.php
* \Zend\Mail\Message => /path/to/project/lib/vendor/Zend/Mail/Message.php

Underscores in Namespaces and Class Names

* \namespace\package\Class\_Name => /path/to/project/lib/vendor/namespace/package/Class/Name.php
* \namespace\package\_name\Class\_Name => /path/to/project/lib/vendor/namespace/package\_name/Class/Name.php

The standards we set here should be the lowest common denominator for painless autoloader interoperability. You can test that you are following these standards by utilizing this sample SplClassLoader implementation which is able to load PHP 5.3 classes.

Example Implementation

Below is an example function to simply demonstrate how the above proposed standards are autoloaded.

<?php

**function** **autoload**($className)

{

$className = ltrim($className, '\\');

$fileName = '';

$namespace = '';

**if** ($lastNsPos = strrpos($className, '\\')) {

$namespace = substr($className, 0, $lastNsPos);

$className = substr($className, $lastNsPos + 1);

$fileName = str\_replace('\\', DIRECTORY\_SEPARATOR, $namespace) . DIRECTORY\_SEPARATOR;

}

$fileName .= str\_replace('\_', DIRECTORY\_SEPARATOR, $className) . '.php';

**require** $fileName;

}

spl\_autoload\_register('autoload');

SplClassLoader Implementation

The following gist is a sample SplClassLoader implementation that can load your classes if you follow the autoloader interoperability standards proposed above. It is the current recommended way to load PHP 5.3 classes that follow

|  |  |  |
| --- | --- | --- |
| PSR-1 | Basic Coding Standard | It comprises what should be considered the standard coding elements that are required to ensure a high level of technical interoperability between shared PHP code.[[5]](https://en.wikipedia.org/wiki/PHP_Standard_Recommendation#cite_note-5) |

* 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](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md), [PSR-4](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md)].
* 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

// 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

// declaration

**function** **foo**()

{

// function body

}

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

**if** (! function\_exists('bar')) {

**function** **bar**()

{

// function body

}

}

## 3. Namespace and Class Names

Namespaces and classes MUST follow an "autoloading" PSR: [[PSR-0](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md), [PSR-4](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md)].

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 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 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

**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().

|  |  |  |
| --- | --- | --- |
| PSR-2 | Coding Style Guide | It considers PSR-1 and it is intended 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.[[6]](https://en.wikipedia.org/wiki/PHP_Standard_Recommendation#cite_note-6) |

* Code MUST follow a "coding style guide" PSR [[PSR-1](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md)].
* 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** **sampleMethod**($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](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md).

### 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](http://php.net/manual/en/reserved.keywords.php) 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

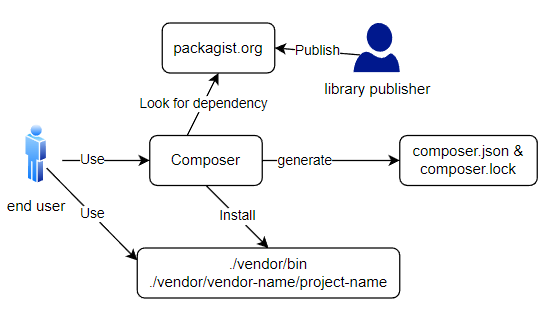
Visibility MUST be declared on all methods.

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

Method names MUST NOT be declare

Composer is a PHP library dependency tool that resolves complicated dependencies between PHP libraries for a project. For Linux users, you might compare it with Yum or Apt. However, there is a fundamental difference between them. Composer is mainly used to main dependencies for a project under a directory, although a global installation is available if explicitly requested.

## How does PHP Composer Work?



**Composer Autoload**

Composer autoloading provides a standard approach in defining what classes will be loaded automatically when needed. Also, it allows to specify what files will be included by the runtime at the very beginning of request or the program start. When the compiler processes the "autoload" section, it remembers the autoloading rules and stores the information in metadata. It does not use the information to compile any additional files - all the files meant to be compiled have to be listed within the <Compile> MSBuild item group explicitly.

* Rules from composer.json will be stored within the assembly metadata; hence the original composer.json file is not required to be bundled with the compiled application.
* Files listed in "autoload" are not automatically added to the compilation. Files needed to be compiled have to be listed within the <Compile> item group explicitly.
* A syntax error in the json file will cause the compilation to stop with a corresponding error message. A wrong value in the composer.json causes a compile-time warning and the compilation will continue.

*Sample "composer.json":*

{

"autoload": {

"psr-4": {

"Vendor\\Namespace\\": "src/",

"": "src/",

},

"psr-0": {

"Vendor\\Namespace\\": "src/",

"Vendor\_Namespace\_": "src/"

},

"classmap": ["src/", "lib/", "Something.php"],

"exclude-from-classmap": ["/src/Tests/", "/lib/tests/"],

"files": [],

}

}

## Magic Constants

Here are the magic constants, with descriptions and examples:

|  |  |  |
| --- | --- | --- |
| **Constant** | **Description** |  |
| \_\_CLASS\_\_ | If used inside a class, the class name is returned. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_class) |
| \_\_DIR\_\_ | The directory of the file. | [Try it »](https://www.w3schools.com/php/phpshowit.php?filename=tryphp_const_dir) |
| \_\_FILE\_\_ | The file name including the full path. | [Try it »](https://www.w3schools.com/php/phpshowit.php?filename=tryphp_const_file) |
| \_\_FUNCTION\_\_ | If inside a function, the function name is returned. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_function) |
| \_\_LINE\_\_ | The current line number. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_line) |
| \_\_METHOD\_\_ | If used inside a function that belongs to a class, both class and function name is returned. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_method) |
| \_\_NAMESPACE\_\_ | If used inside a namespace, the name of the namespace is returned. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_namespace) |
| \_\_TRAIT\_\_ | If used inside a trait, the trait name is returned. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_trait) |
| ClassName::class | Returns the name of the specified class and the name of the namespace, if any. | [Try it »](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_const_classname) |

**Note:**

The magic constants are case-insensitive, meaning \_\_LINE\_\_ returns the same as \_\_line\_\_.