We use coding standard so that a programmer can understand the code you produce. A coding standard acts as the blueprint for all the team to decipher the code.

1.Php coding standard :

Introduction:

There are multiple ways to complete a project.The language ,code style can vary.A massive variety in options,leads to wildly different codebases

Indentation :

Spacing and indentation should be consistent throughout your code. Many developers choose to use 4-space or 2-space indentation. In PHP, each nested statement should be indented exactly once more than the previous line's indentation.

1 Exam:

```
1.function for {
2.$x = 2;
3.print($x);
4.}
5.
6.function bar() {
7. print(1);
8.}
```

Line Formatting:

Opening "{" brackets should be on the same line as the function/loop/conditional header, and corresponding closing "}" brackets should be on their own line

}

spacing

Place a space between operators, assigments ("=") and their operands

Exam: `$x = ($a + $b) * $c / $d + foo();`

Control Structures:

These include if, for, while, switch, etc. Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls**.**

Exam:

```
if ((condition1) || (condition2)) {
   action1;
}elseif ((condition3) && (condition4)) {
   action2;
}else {
   default action;
}
```

Function Calls:

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter

---

Exam: `$var = foo($bar, $baz, $quux);`

Function definitions :

```
function fooFunction($arg1, $arg2 = '') {
    if (condition) {
        statement;
    }
    return $val;
}
```

Comments:

C style comments (/* */) and standard C++ comments (//) are both fine. Use of Perl/shell style comments (#) is discouraged.

PHP Code Tags :

Always use <?php ?> to delimit PHP code, not the <? ?> shorthand. This is required for PHP compliance and is also the most portable way to include PHP code on differing operating systems and setups.

Variable Names :

- Use all lower case letters
- Use '_' as the word separator.
- Global variables should be prepended with a 'g'.
- Global constants should be all caps with '_' separators.
- Static variables may be prepended with 's'.

Make Functions Reentrant :

Functions should not keep static variables that prevent a function from being reentrant.

Alignment of Declaration Blocks: Block of declarations should be aligned.

One Statement Per Line:

There should be only one statement per line unless the statements are very closely related.

Short Methods or Functions:

Methods should limit themselves to a single page of code.

2. This document describes PHP coding standards for Phabricator and related projects. This document outlines technical and style guidelines which are followed in libphutil. Contributors should also follow these guidelines.22

Linebreaks and Indentation: 2

- Use two spaces for indentation. Don't use tab literal characters.
- Use Unix linebreaks ("\n"), ("\r\n") or OS9 ("\r").
- Put a space after control keywords like `if` and `for`.
- Put a space after commas in argument lists.
- Put a space around operators like =, <, etc.
- Don't put spaces after function names.
- Parentheses should hug their contents.
- Generally, prefer to wrap code at 80 columns.

Capitalization:

- Name variables and functions using `lowercase_with_underscores`.
- Name classes using `UpperCamelCase`.
- Name methods and properties using `lowerCamelCase`.
- Use uppercase for common acronyms like ID and HTML.
- Name constants using `UPPERCASE`.
- Write `true`, `false` and `null` in lowercase.

PHP Language Style:

- Use "<?php", not the "<?" short form. Omit the closing "?>" tag.
- Prefer casts like `(string)` to casting functions like `strval()`.
- Prefer type checks like `$v === null` to type functions like `is_null()`.
- Avoid all crazy alternate forms of language constructs like "endwhile" and "<>".
- Always put braces around conditional and loop blocks.

Examples:

if/else:

```
if ($some_variable > 3) {

  // ...

} else if ($some_variable === null) {

  // ...

} else {

  // ...
}
```

for:

```
for ($ii = 0; $ii < 10; $ii++) {

  // ...

}
```

switch:

```
switch ($value) {

  case 1:

    // ...

    break;
  case 2:
    if ($flag) {

      // ...

      break;
    }
```

```
    break;
  default:

    // ...

    break;
}
```

method definitions:

```
function example_function($base_value,
$additional_value) {
  return $base_value + $additional_value;
}

class C {
  public static function
promulgateConflagrationInstance(
    IFuel $fuel,
    IgnitionSource $source) {

    // ...
  }
}
```

class:

```
class Dog extends Animal {

  const CIRCLES_REQUIRED_TO_LIE_DOWN = 3;

  private $favoriteFood = 'dirt';

  public function getFavoriteFood() {

    return $this->favoriteFood;
  }
}
```

3.php coding style

PSR-2 is an extension of the PSR-1 coding standard. Some examples of its contents are:

- You must follow PSR-1 coding standards
- 4 spaces must be used for indents. Using tabs is not allowed
- There is no limit to line length, but it should be under 120 characters, and best if under 80
- You must put a newline before curly braces for classes and methods
- Methods and properties must be defined with `abstract`/`final` first, followed with `public`/`protected`, and finally `static`.
- You must not put a newline before curly braces in conditional statements
- You must not put any spaces before `(` and `)` in conditional statements

CakePHP and Symfony, which will be explained later on in this article, are based on this PSR-2 standard.

Defining Classes

You must put a newline before `{` in class definitions.
Also, `extends` and `implements` must be written on the same line as the class name.

```
class ClassName extends ParentClassName implements Interface1,
Interface2
{
    // Class definition
}
```

If there are too many interfaces for one line, you should put a newline after `implements` and write one interface per line like below.

```
class ClassName extends ParentClassName implements
    Interface1,
    Interface2,
    Interface3,
    Interface4
{
    // Class definition
}
```

Since there are quite a few standards that recommend writing `{` on the same line like `class ClassName {`, this may be a style which you haven't seen before.

Defining Properties

In PSR-2, you must not omit `public`/`protected`/`private` modifiers. In PHP, properties become `public` if these are omitted, but because it is hard to tell if one purposely omitted these modifiers or they just forgot, you should always explicitly write `public`. The `static` keyword comes next. You must not use `var` when defining properties because you can't add any modifiers to `var`.

```
class ClassName
{
    public $property1;
    private $property2;
```

```
    public static $staticProperty;
}
```

Additionally, you must not define two or more properties with one statement. You *can* define properties in the way shown below but it is prohibited in PSR-2.

```
class ClassName
{
     private $property1, $property2;
}
```

Methods

Like properties, you must have either one of `public`/`protected`/`private` and `abstract`/`final` comes after them if used. `static` is the last modifier. You must not put any spaces before and after braces, and you must put a newline before curly braces. Also, you must not put any whitespaces before commas in arguments, and you must put one whitespace after them.

```
class ClassName
{
    abstract protected function abstractDoSomething();    final
public static function doSomething($arg1, $arg2, $arg3)
    {
        // ...
    }
}
```

If there are too many arguments, you can put a newline after `(` and write one argument per line. In this case, you can't write multiple variables on one line. Also, you should write `)` and `{` on the same line, separated by a whitespace.

```
class ClassName
{
    public function doSomething(
        TypeHint $arg1,
        $arg2,
        $arg3,
        $arg4
    ) {
```

```
        // ...
    }
}
```

Please note that you must not put a newline before `{` in closures.

```
$closure = function ($a, $b) use ($c) {
    // Body
};
```

Conditional Statements

For conditional statements,

- You must put one whitespace before `(`
- You must not put any whitespaces after `(`
- You must not put any whitespaces before `)`
- You must put one whitespace after `)`

Also, use `elseif` rather than `else if`.

```
if ($condition1) {
    // ...
} elseif ($condition2) {
    // ...
} else {
    // ...
}
```

Be careful, `else if` and `elseif` are not the complete same things. `elseif` is one statement by itself, but `else if` on the other hand is interpreted as an `if` statement in the `else` of the first `if`.

```
if ($condition1) {
    // ...
} else if ($condition2) {
    // ...
} else {
```

```
    // ...
}
```

The syntax above is actually interpreted like below:

```
if ($condition1) {
    // ...
} else {
    if ($condition2) {
        // ...
    } else {
        // ...
    }
}
```

For `switch` statements, `case` statements must be indented once from `switch`, and bodies for the `case`s must be indented once from `case`. When not `break`ing after any kind of operations in `case`, you must write a comment.

```
switch ($condition) {
    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,
```

4. Symfony Coding Standard

About Symfony

Symfony is another fairly old open source web framework, like CakePHP. Similarly, the developers participate in PHP-FIG, so the standards again basically follow PSR-2 and have a few additional standards. However, the additional standards follow a slightly different mindset.

Notations for Use Yoda Comparisons

Using Yoda notations is recommended when using comparisons with `==`, `===`, `!=`, `!==`, etc.. This is the complete opposite from CakePHP.

```
// recommended (called Yoda notations, or Yoda conditions)
if (null === $value) {
    // ...
}// not recommended (was recommended in CakePHP)
if ($value === null) {
    // ...
}
```

As stated in the CakePHP chapter, writing comparisons like `$value === null` do not result in an error when mistaking `===` with `=`, and is hard to notice such bugs. Therefore, Yoda notations are recommended in Symfony.

Put a Comma after the Last Element of Arrays Taking Multiple Lines

The following two styles of writing arrays result in the same values. However, the first style is recommended.

```
// recommended style
$array = [
    'value1',
    'value2',
    'value3', // comma at the end of last line
];// not recommended
$array = [
    'value1',
    'value2',
    'value3'  // no comma at the last line
];
```

This is because it is easier to read the differences when adding elements.

```
// recommended
 $array = [
     'value1',
     'value2',
     'value3', // comma at the end of last line
+    'value4', // only this line is shown as the difference
 ]; // not recommended
 $array = [
     'value1',
     'value2',
-    'value3'  // no comma at the last line
+    'value3', // no comma at the last line
+    'value4'  // the line above is also shown as the difference
 ];
```

With the unrecommended style, one more line shows up as the difference since you have to add a comma to the former last line. If an array takes multiple lines, you should also put a comma after the last element.

Write Arguments on the Same Line as Method or Function Names

You should write arguments for methods and functions on the same line as the method or function names.

```
public function doSomething($arg1, $arg2, $longNameArgument3,
$longNameArgument4, $longNameArgument5)
{
    // ...
}
```

This rule is considered to be intended to restrain method and function definitions with long arguments and to increase code legibility by making it

Taking Multiple Lines

The following two styles of writing arrays result in the same values. However, the first style is recommended.

```
// recommended style
$array = [
    'value1',
    'value2',
    'value3', // comma at the end of last line
];// not recommended
$array = [
    'value1',
    'value2',
    'value3'  // no comma at the last line
];
```

This is because it is easier to read the differences when adding elements.

```
// recommended
 $array = [
     'value1',
     'value2',
     'value3', // comma at the end of last line
```

```
+    'value4', // only this line is shown as the difference
 ]; // not recommended
 $array = [
     'value1',
     'value2',
-    'value3'  // no comma at the last line
+    'value3', // no comma at the last line
+    'value4'  // the line above is also shown as the difference
 ];
```

With the unrecommended style, one more line shows up as the difference since you have to add a comma to the former last line. If an array takes multiple lines, you should also put a comma after the last element.

Write Arguments on the Same Line as Method or Function Names

You should write arguments for methods and functions on the same line as the method or function names.

```
public function doSomething($arg1, $arg2, $longNameArgument3,
$longNameArgument4, $longNameArgument5)
{
    // ...
}
```

This rule is considered to be intended to restrain method and function definitions with long arguments and to increase code legibility by making it easier to distinguish the function's arguments and body. However, this also results in longer lines. This may be a rule that goes a different way from CakePHP, which restricts the length of a line.

The Order of Properties/Methods in a Class

In a class, properties are defined first, and then the methods. For the order of properties and methods, public methods come first, then protected, and finally private. Among the methods, constructors (`__construct()`), PHPUnit, `setUp()` and `tearDown()` are defined first.

```
class Something
{
    public $property1;    protected $property2;    private
$property3;    public function __construct()
    {
        // ...
    }    public function doSomething()
    {
        // ...
    }    private function doSomethingPrivate()
    {
        // ...
    }
}
```

1.HTML coding standard

Introduction:

HTML stands for Hyper Text Markup Language and describes the structure of a Web page. HTML consists of a series of elements.HTML elements tell the browser how to display the content,HTML elements are represented by tags HTML tags label pieces of content such as "heading", ."paragraph", "table", and so on Browsers do not display the HTML tags, but use them to render the content of the page.

Formatting

All HTML documents must use **two spaces** for indentation and there should be no trailing whitespace. HTML5 syntax must be used and all attributes must use double quotes around attributes.

Doctype and layout

All documents must be using the HTML5 doctype and the `<html>` element should have a `"lang"` attribute. The `<head>` should also at a minimum include `"viewport"` and `"charset"` meta tags.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Example Site</title>
  </head>
  <body></body>
</html>
```

Forms

Form fields must always include a `<label>` element with a `"for"` attribute matching the `"id"` on the input. This helps accessibility by focusing the input when the label is clicked, it also helps screen readers match labels to their respective inputs.

```
<label for="field-email">email</label>
<input type="email" id="field-email" name="email" value="" />
```

Each `<input>` should have an `"id"` that is unique to the page. It does not have to match the `"name"` attribute.

Forms should take advantage of the new HTML5 input types where they make sense to do so, placeholder attributes should also be included where relevant. Including these can provided enhancements in browsers that support them such as tailored inputs and keyboards.

```
<div>
  <label for="field-email">Email</label>
  <input type="email" id="field-email" name="email" value="name@example.com">
</div>
<div>

  <label for="field-phone">Phone</label>
  <input type="phone" id="field-phone" name="phone" value="" placeholder="+44 077 12345 678">
</div>
<div>
  <label for="field-url">Homepage</label>
  <input type="url" id="field-url" name="url" value="" placeholder="http://example.com">

</div>
```

Including meta data

Classes should ideally only be used as styling hooks. If you need to include additional data in the HTML document, for example to pass data to JavaScript, then the HTML5 `data-` attributes should be used.

```
<a class="btn" data-format="csv">Download CSV</a>
```

These can then be accessed easily via jQuery using the `.data()` method.

```
jQuery('.btn').data('format'); //=> "csv"

// Get the contents of all data attributes.
jQuery('.btn').data(); => {format: "csv"}
```

One thing to note is that the JavaScript API for datasets will convert all attribute names into camelCase. So `"data-file-format"` will become `fileFormat`.

For example:

```
<a class="btn" data-file-format="csv">Download CSV</a>
```

Will become:

```
jQuery('.btn').data('fileFormat'); //=> "csv"
jQuery('.btn').data(); => {fileFormat: "csv"}
```

2.HTML coding standard

HTML Text Formatting:

HTML also defines special **elements** for defining text with a special **meaning**.

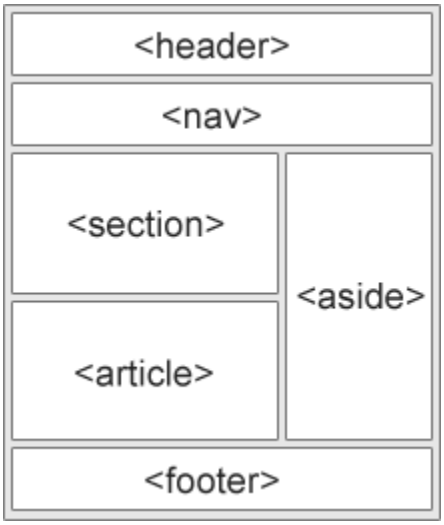
HTML uses elements like `<b>` and `<i>` for formatting output, like **bold** or *italic* text.

Formatting elements were designed to display special types of text:

- `<b>` - Bold text
- `<strong>` - Important text
- `<i>` - Italic text
- `<em>` - Emphasized text
- `<mark>` - Marked text
- `<small>` - Small text
- `<del>` - Deleted text
- `<ins>` - Inserted text
- `<sub>` - Subscript text
- `<sup>` - Superscript text

HTML Layouts : Websites often display content in multiple columns (like a magazine or newspaper).

HTML offers several semantic elements that define the different parts of a web page:



- <header> - Defines a header for a document or a section
- <nav> - Defines a container for navigation links
- <section> - Defines a section in a document
- <article> - Defines an independent self-contained article
- <aside> - Defines content aside from the content (like a sidebar)
- <footer> - Defines a footer for a document or a section
- <details> - Defines additional details
-
- <summary> - Defines a heading for the <details> element.

HTML coding document :

<!DOCTYPE html>

<html>

<body>

<h1>My First Heading</h1>

My first paragraph.

</body>

</html>

HTML Comment Tags

You can add comments to your HTML source by using the following syntax:

```
<!-- Write your comments here -->
```

```
abstract class AbstractDatabase
{
    // ...
}
```

You should add an `Interface` suffix to an interface.

```
interface LoggerInterface
{
    // ...
}
```

You should add a `Trait` suffix to a trait.

```
trait SomethingTrait
{
    // ...
}
```

You should add an `Exception` suffix to an exception.

```
class NotFoundException extends \RuntimeException
{
    // ...
}
```