PHP CODING GUIDELINE

==========

Basic Coding Standard :

Follow PSR-0 & PSR-2 standard for coding style.

The following two files must be included inside docs folder.

1.Psr1.pdf [https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md]

2.Psr2.pdf [http://www.php-fig.org/psr/psr-2/]

PHP CODE QUALITY CHECKER TOOL

==========

1. Use PHPCI for checking code quality. [phptesting.org]
2. Your PHPCI build must be passed before any release.

Optionally you can also use **Sonarqube. [http://www.sonarqube.org/]**

Rules :

The following points should be taken care of while writing php codes.

1. "$this" should not be used in a static context

Desc :

$this refers to the current class instance. But static methods can be accessed without instantiating the class, and $this is not available to them. Using $this in a static context will result in runtime errors.

Noncompliant Code Example

```
class Clazz {
  $name=NULL;  // instance variable
  public static function foo(){
    if ($this->name != NULL) {
      // ...
    }
  }
}
```

Compliant Solution

```
class Clazz {
  $name=NULL;  // instance variable

  public static function foo($nameParam){
    if ($nameParam != NULL) {
      // ...
    }
  }
}
```

2. <?php tag should be used and <?= should not be used.

3. "exit(...)" and "die(...)" statements should not be used.
Desc :
The exit(...) and die(...) statements should absolutely not be used in Web PHP pages as this might lead to a very bad user experience. In such case, the end user might have the feeling that the web site is down or has encountered a fatal error.

But of course PHP can also be used to develop command line application and in such case use of exit(...) or die(...) statement can be justified but must remain limited and not spread all over the application. We expect exceptions to be used to handle errors and those exceptions should be caught just before leaving the application to specify the exit code with help of exit(...) or die(...) statements.

Noncompliant Code Example

```
class Foo {
    public function bar($param)  {
        if ($param === 42) {
            exit(23);
        }
    }
}
```

Compliant Solution

```
class Foo {
    public function bar($param)  {
        if ($param === 42) {
            throw new Exception('Value 42 is not expected.');
        }
    }
}
```

4. "final" should not be used redundantly.
Desc :
There is no need to use the final modifier inside a final class. Everything in it is final by default.

Noncompliant Code Example

```
final class MyClass {

  public final String getName() {  // Noncompliant
    return name;
  }
}
```

Compliant Solution

```
final class MyClass {

  public String getName() {  // Compliant
    return name;
  }
}
```

5. "FIXME" tags should be handled.
Desc :
FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later.
Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags, and ensure that they do not go unnoticed.

Noncompliant Code Example

```
function divide($numerator, $denominator) {
  return $numerator / $denominator;              // FIXME denominator value
might be  0
}
```

6. "for" loop stop conditions should be invariant.
Desc :
for loop stop conditions must be invariant (i.e. true at both the beginning and ending of every loop iteration). Ideally, this means that the stop condition is set to a local variable just before the loop begins.

Stop conditions that are not invariant are difficult to understand and maintain, and will likely lead to the introduction of errors in the future.

This rule tracks three types of non-invariant stop conditions:

When the loop counters are updated in the body of the for loop
When the stop condition depend upon a method call
When the stop condition depends on an object property, since such properties could change during the execution of the loop.
Noncompliant Code Example

```
for ($i = 0; $i < 10; $i++) {
  echo $i;
  if(condition) {
    $i = 20;
  }
}
```

Compliant Solution

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

7. "goto" statement should not be used.
Desc :
goto is an unstructured control flow statement. It makes code less readable and maintainable. Structured control flow statements such as if, for, while, continue or break should be used instead.

Noncompliant Code Example

```
$i = 0;
loop:
  echo("i = $i");
  $i++;
  if ($i < 10){
    goto loop;
  }
```

Compliant Solution

```
for ($i = 0; $i < 10; $i++){
  echo("i = $i");
}
```

8. "if ... else if" constructs shall be terminated with an "else" clause.
Desc :
This rule applies whenever an if statement is followed by one or more else if statements; the final else if shall be followed by an else statement.

The requirement for a final else statement is defensive programming. The else statement should either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final default clause in a switch statement.

Noncompliant Code Example

```
if (condition1) {
  do_something();
} else if (condition2) {
  do_something_else();
}
```

Compliant Solution

```
if (condition1) {
  do_something();
} else if (condition2) {
  do_something_else();
} else {
  throw new InvalidArgumentException('message');
}
```

9. "php_sapi_name()" should not be used.
Desc :
Both php_sapi_name() and the PHP_SAPI constant give the same value. But calling the method is less efficient that simply referencing the constant.

Noncompliant Code Example

```
if (php_sapi_name() == 'test') { ... }
```

Compliant Solution

```
if (PHP_SAPI == 'test') { ... }
```

10 . "require_once" and "include_once" should be used instead of "require" and “include”.
Desc :
At root, require, require_once, include, and include_once all perform the same task of including one file in another. However, the way they perform that task differs, and they should not be used interchangeably.

require includes a file but generates a fatal error if an error occurs in the process.

include also includes a file, but generates only a warning if an error occurs.

Predictably, the difference between require and require_once is the same as the difference between include and include_once - the "_once" versions ensure that the specified file is only included once.

Because including the same file multiple times could have unpredictable results, the "once" versions are preferred.

Because include_once generates only warnings, it should be used only when the file is being included conditionally, i.e. when all possible error conditions have been checked beforehand.

Noncompliant Code Example

```
include 'code.php';  //Noncompliant; not a "_once" usage and not conditional
include $user.'_history.php'; // Noncompliant
require 'more_code.php';  // Noncompliant; not a "_once" usage
```

Compliant Solution

```
require_once 'code.php';
if (is_member($user)) {
  include_once $user.'_history.php';
}
require_once ‘more_code.php';
```

11. "sleep" should not be called.

Desc :

sleep is sometimes used in a mistaken attempt to prevent Denial of Service (DoS) attacks by throttling response rate. But because it ties up a thread, each request takes longer to serve that it otherwise would, making the application more vulnerable to DoS attacks, rather than less.

Noncompliant Code Example

```
if (is_bad_ip($requester)) {
  sleep(5);  // Noncompliant
}
```

12. "switch case" clauses should not have too many lines.

Desc :

The switch statement should be used only to clearly define some new branches in the control flow. As soon as a case clause contains too many statements this highly decreases the readability of the overall control flow statement. In such case, the content of case clause should be extracted in a dedicated function

Noncompliant Code Example

The following code snippet illustrates this rule with the default threshold of 5:

```
switch ($var) {
  case 0:  // 6 lines till next case
    methodCall1();
    methodCall2();
    methodCall3();
    methodCall4();
    break;
  default:
    break;
}
```

Compliant Solution

```
switch ($var) {
  case 0:
    doSomething();
    break;
  default:
    break;
}

function doSomething(){
  methodCall1("");
  methodCall2("");
  methodCall3("");
  methodCall4("");
}
```

13. "switch" statements should end with a "case default" clause.
Desc :
The requirement for a final [defaultlOTHERS] clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken. Even when the switch covers all current values of an enum, a default case should still be used because there is no guarantee that the enum won't be extended.

Noncompliant Code Example

```
switch ($param) {  //missing default clause
  case 0:
    do_something();
    break;
  case 1:
    do_something_else();
    break;
}

switch ($param) {
  default: // default clause should be the last one
    error();
    break;
  case 0:
    do_something();
    break;
  case 1:
    do_something_else();
    break;
}
```

Compliant Solution

```
switch ($param) {
  case 0:
    do_something();
    break;
  case 1:
    do_something_else();
    break;
  default:
    error();
    break;
}
```

14. "switch" statements should have at least 3 "case" clauses.
Desc :
switch statements are useful when there are many different cases depending on the value of the same expression.
For just one or two cases however, the code will be more readable with if statements.

Noncompliant Code Example

```
switch ($variable) {
  case 0:
    do_something();
    break;
  default:
    do_something_else();
    break;
}
```

Compliant Solution

```
if ($variable == 0) {
  do_something();
} else {
  do_something_else();
}
```

15. "switch" statements should not have too many "case" clauses.
Desc :
When switch statements have a large set of case clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead.

16. "TODO" tags should be handled.
Desc :
TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later.
Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags, and ensure that they do not go unnoticed.

Noncompliant Code Example

```
function doSomething() {
  // TODO
}
```

17. A close curly brace should be located at the beginning of a line.
Desc :
Shared coding conventions make it possible for a team to efficiently collaborate. This rule makes it mandatory to place a close curly brace at the beginning of a line.

Noncompliant Code Example

```
if(condition) {
  doSomething();}
```

Compliant Solution

```
if(condition) {
  doSomething();
}
```

Exceptions

When blocks are inlined (open and close curly braces on the same line), no issue is triggered.

```
if(condition) {doSomething();}
```

18. Class constructors should not create other objects.
Desc :
Dependency injection is a software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object (or client) and are made part of the client's state. The pattern separates the creation of a client's dependencies from its own behavior, which allows program designs to be loosely coupled and to follow the dependency inversion and single responsibility principles.

Noncompliant Code Example

```
class SomeClass {

  public function __construct() {
    $this->object = new SomeOtherClass();
  }
}
```

Compliant Solution

```
class SomeClass {

  public function __construct(SomeOtherClass $object) {
    $this->object = $object;
  }
}
```

19. Classes should not be coupled to too many other classes (Single Responsibility Principle).
Desc:
According to the Single Responsibility Principle, introduced by Robert C. Martin in his book "Principles of Object Oriented Design", a class should have only one responsibility:

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.
Classes which rely on many other classes tend to aggregate too many responsibilities and should be split into several smaller ones.
Nested classes dependencies are not counted as dependencies of the outer class.

20. Classes should not be too complex.
Desc :
The cyclomatic complexity of a class should not exceed a defined threshold. Complex code can perform poorly and will in any case be difficult to understand and therefore to maintain.

21. Classes should not have too many fields.
Desc :
A class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain, and having a lot of fields is an indication that a class has grown too large.

Above a specific threshold, it is strongly advised to refactor the class into smaller ones which focus on well defined topics.

22. Classes should not have too many lines.
Desc :
A class that grows too much tends to aggregate too many responsibilities, and inevitably becomes harder to understand and to maintain. Above a specific threshold, it is strongly advised to refactor the class into smaller ones which focus on well-defined topics.

23.Classes should not have too many methods.
Desc :
A class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain. Above a specific threshold, it is strongly advised to refactor the class into smaller ones which focus on well defined topics.

24. Closing tag "?>" should be omitted on files containing only PHP.
Desc :
If a file is pure PHP code, it is preferable to omit the PHP closing tag at the end of the file. This prevents accidental whitespace or new lines being added after the PHP closing tag, which may cause unwanted effects because PHP will start output buffering when there is no intention from the programmer to send any output at that point in the script.

25. Code should not be dynamically injected and executed to prevent Eval Injection vulnerability.
Desc:
The eval function is a way to run arbitrary code at run-time.

According to the PHP documentation

The eval() language construct is very dangerous because it allows execution of arbitrary PHP code. Its use thus is discouraged. If you have carefully verified that there is no other option than to use this construct, pay special attention not to pass any user provided data into it without properly validating it beforehand.
Noncompliant Code Example

```
eval($code_to_be_dynamically_executed)
```

26. Collapsible "if" statements should be merged.
Desc:
Merging collapsible if statements increases the code's readability.

Noncompliant Code Example

```
if (condition1) {
  if (condition2) {
    ...
  }
}
```

Compliant Solution

```
if (condition1 && condition2) {
  ...
}
```

27. Colors should be defined in upper case.
Desc:
Shared coding conventions allow teams to collaborate effectively. Writing colors in upper case makes them stand out at such, thereby making the code easier to read.

This rule checks that hexadecimal color definitions are written in upper case.

Noncompliant Code Example

```
$white = '#ffffff';  // Noncompliant
$dkgray = '#006400';
$aqua = '#00ffff';  // Noncompliant
```

Compliant Solution

```
$white = '#FFFFFF';  // Compliant
$dkgray = '#006400';
$aqua = '#00FFFF';  // Compliant
```

28. Comments should not be located at the end of lines of code.
Desc:
This rule verifies that single-line comments are not located at the end of a line of code. The main idea behind this rule is that in order to be really readable, trailing comments would have to be properly written and formatted (correct alignment, no interference with the visual structure of the code, not too long to be visible) but most often, automatic code formatters would not handle this correctly: the code would end up less readable. Comments are far better placed on the previous empty line of code, where they will always be visible and properly formatted.

Noncompliant Code Example

```
$a = $b + $c; // This is a trailing comment that can be very very long
```

Compliant Solution

```
// This very long comment is better placed before the line of code
$a = $b + $c;
```

Exceptions

By default, the property "legalTrailingCommentPattern" allows to ignore comments containing only one word :

```
doSomething(); //FIXME
```

29. Configuration should not be changed dynamically.
Desc:
ini_set changes the value of the given configuration option for the duration of the script's execution. While there may be a reason to do this, you should make sure that it's a very good reason indeed, because this is the sort of "magic" change which can cause severe teeth-gnashing and hair tearing when the script needs to be debugged.

For instance, if the user explicitly turns logging on for a script, but then the script itself uses ini_set('display_errors', 0); to turn logging back off, it is likely that every other aspect of the environment will be examined before, in desperation, the script is read to figure out where the logging is going.

Noncompliant Code Example

```
ini_set('display_errors', 0);  // Noncompliant
```

30. Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply.
Desc:
Nested if, for, while, switch and try statements is a key ingredient for making what's known as "Spaghetti code".

Such code is hard to read, refactor and therefore maintain.

Noncompliant Code Example

The following code snippet illustrates this rule with the default threshold of 3.

```
  if (condition1) {               // Compliant - depth = 1
    ...
    if (condition2) {             // Compliant - depth = 2
      ...
      for($ = 0; $i < 10; $i++) {  // Compliant - depth = 3, not exceeding the limit
        ...
        if (condition4) {         // Non-Compliant - depth = 4
          if (condition5) {       // Depth = 5, exceeding the limit, but issues are
only reported on depth = 4
            ...
          }
          return;
        }
      }
    }
  }
```

31. Credentials should not be hard-coded.
Desc :
Because it is easy to extract strings from a compiled application, credentials should never be hard-coded. Do so, and they're almost guaranteed to end up in the hands of an attacker. This is particularly true for applications that are distributed.

Credentials should be stored outside of the code in a strongly-protected encrypted configuration file or database.

Noncompliant Code Example

```
$uname = "steve";
$password = "blue";
connect($uname, $password);
```
Compliant Solution

```
$uname = getEncryptedUser();
$password = getEncryptedPass();
connect($uname, $password);
```

33. Deprecated predefined variables should not be used.
Desc:
The following predefined variables are deprecated and should be replaced by the new versions:

Replace	With
$HTTP_SERVER_VARS	$_SERVER
$HTTP_GET_VARS	$_GET
$HTTP_POST_VARS	$_POST
$HTTP_POST_FILES	$_FILES
$HTTP_SESSION_VARS	$_SESSION
$HTTP_ENV_VARS	$_ENV
$HTTP_COOKIE_VARS	$_COOKIE

Noncompliant Code Example

```
echo 'Name parameter value: ' . $HTTP_GET_VARS["name"];
```

Compliant Solution

```
echo 'Name parameter value: ' . $_GET[“name"];
```

34. Errors should not be silenced.
Desc:
Just as pain is your body's way of telling you something is wrong, errors are PHP's way of telling you there's something you need to fix. Neither pain, nor PHP errors should be ignored.

Noncompliant Code Example

```
@doSomethingDangerous($password);  // Noncompliant; '@' silences errors from function call
```

Compliant Solution

```
doSomethingDangerous($password);
```

35. Literal boolean values should not be used in condition expressions.
Desc:
Remove literal boolean values from conditional expressions to improve readability. Anything that can be tested for equality with a boolean value must itself be a boolean value, and boolean values can be tested atomically.

Noncompliant Code Example

```
if ($booleanVariable == true) { /* ... */ }
if ($booleanVariable != true) { /* ... */ }
if ($booleanVariable || false) { /* ... */ }
doSomething(!false);

$booleanVariable = condition ? true : exp;
$booleanVariable = condition ? false : exp;
$booleanVariable = condition ?  exp : true;
$booleanVariable = condition ?  exp : false;
```

Compliant Solution

```
if ($booleanVariable) { /* ... */ }
if (!$booleanVariable) { /* ... */ }
if ($booleanVariable) { /* ... */ }
doSomething(true);

$booleanVariable = condition || exp;
$booleanVariable = !condition && exp;
$booleanVariable = !condition ||  exp;
$booleanVariable = condition && exp;
```

Exceptions

The use of literal booleans in comparisons which use identity operators (=== and !==) are ignored.

36. Functions deprecated in PHP 5 should not be used.
Desc:
Deprecated language features are those that have been retained temporarily for backward compatibility, but which will eventually be removed from the language. In effect, deprecation announces a grace period to allow the smooth transition from the old features to the new ones. In that period, no use of the deprecated features should be added to the code, and all existing uses should be gradually removed.

The following functions were deprecated in PHP 5:

Deprecated	Use Instead
call_user_method()	call_user_func()
call_user_method_array()	call_user_func_array()
define_syslog_variables()	
dl()	
ereg()	preg_match()
ereg_replace()	preg_replace()
eregi()	preg_match() with 'i' modifier
eregi_replace()	preg_replace() with 'i' modifier
set_magic_quotes_runtime()	and its alias, magic_quotes_runtime()
session_register()	$_SESSION superglobal
session_unregister()	$_SESSION superglobal
session_is_registered()	$_SESSION superglobal
set_socket_blocking()	stream_set_blocking()
split()	preg_split()
spliti()	preg_split() with 'i' modifier
sql_regcase()	
mysql_db_query()	mysql_select_db() and mysql_query()
mysql_escape_string()	mysql_real_escape_string()
Passing locale category names as strings	Use the LC_* family of constants

37. Unused code should be deleted.

38. Variable variables should not be used;
Desc:
PHP's variable variables feature is temptingly powerful, but can lead to unmaintainable code.

Noncompliant Code Example

```
$var = 'foo';
$$var = 'bar';      //Noncompliant
$$$var = 'hello';  //Noncompliant

echo $foo; //will display 'bar'
echo $bar; //will display ‘hello'
```

Note :
Add new rules periodically as per your research.

File Naming Convention
==========
Core PHP Project :
==========
1. FileNames will be Pascal cased. [upper camelcase]

CodeIgniter
==========
1. Core files like config, helpers will have their traditional naming convention. filenames in lowercase separated by underscore.

2. If you are going to create a new file inside controller/model/library , then make it pascal case. [Upper camel case]

Laravel
==========
1. File names will be pascal cased. [Upper camel cased]

Folder Naming convention
==========
Core PHP
==========
1. Folder names will be Pascal cased. [Upper camel case]

CodeIgniter
==========
1. Folder names will be in lowercase being separated by underscore.

Laravel
==========
1. Folder names will be in PascalCased.
But It depends upon situation where you need to match namespace with directory name.

Autoload Standard [If followed]
==========
Follow PSR-4 autoload standard if you are auto loading all files through composer.