PHP

General Style and Coding Conventions

May 30, 2022

Revision: 0.21

4050 Esplanade Way

Tallahassee, Florida 32399-0950

Ron DeSantis, Governor

J. Todd Inman, Secretary

Table of Contents

1. Revision History

Date	Revision	Author	Document Changes
5/30/2022	0.23	Stephen Mitchell	Initial Draft
Date			
Date			

Table 1: Revision History

2. Approvals

Role	Name	Title	Signature	Date
Project Sponsor:	Raghib Quresh	Utility System/Engineering Specialist		Date
Project Manager:	Rhonda Ballew	Service Delivery		Date
System Engineer:	Nelson Blocker	Service Delivery		Date

Table 2: Approvals

3. Points of Contact

Name	Title	Email	Contact Number
Rhonda Ballew	Service Delivery	rhonda.ballew@dms.fl.gov	(850) 922-7483
Nelson Blocker	Service Delivery	nelson.blocker@dms.fl.gov	(850) 921-6734
Raghib Quresh	System/Engineering Specialist	raghib.qureshi@dms.fl.gov	(850) 591-0260
Stephen Mitchell	Technical Writer	stephen.mitchell@dms.fl.gov	
DMS SUNCOM Helpdesk			(888) 478-6266

Table 3: Points of Contact

4. Related Documents

Related Code Standard Documents
C — General Style and Coding Conventions
CSS — General Style and Coding Conventions
HTML — General Style and Coding Conventions
JavaScript — General Style and Coding
PL/SQL — General Style and Coding Conventions
SQL — General Style and Coding Conventions
Typescript — General Style and Coding Conventions

Table 4: Related Documents

5. Introduction

What is the need for a PHP coding and style standard? There are many reasons we care about writing clean code and following conventions, such as helping us more easily collaborate and reducing the chances of bugs occurring. In addition, new developers joining the team are expected to produce clean code by adhering to a standard.

Ideally, one should write PHP code that follows a known standard. This could be any combination of PSRs, or one of the coding standards produced by PEAR, Zend, WordPress, and many others. This means other developers can easily read and work with the project's code and applications and have consistency even when working with lots of third-party code.

So, it is no surprise that the Telecommunications Division of Florida's Department of Management Services adopted a standard for the usual reasons; the department and software initiatives are continually growing. Thus, we need to ensure that the efforts remain consistent between all of our software development commitments and ventures.

The visual style of programming code is very important. With the wealth of varied software projects DivTel is engaged in and manages, we want our community of architects and programmers to contribute. This will help us to:

- Read and understand each other's code and consequently spot security problems and opportunities for optimization
- It is a signal of consistency and cleanliness, which is a motivating factor for programmers striving for excellence

To recap, good PHP code should be homogeneous. Whether that means setting rules for the names of variables and functions, adopting standard approaches to recurring tasks like database access and error handling, or simply making sure all of the code is indented the same way, consistency makes your code easier for others to read.

This documentation uses strict language with words such as MUST, MUST NOT, and SHOULD in all caps. Why? Because the document follows the RFC 2119 (Internet Engineering Task Force) of the IETF.

5.1. FILES

The content of this section describes how PHP files are formatted and named.

1. All PHP files MUST use line termination following the Unix text file convention. Lines must end with a single linefeed `LF` character. Linefeed characters are represented as ordinal `10` or hexadecimal `0x0A`.

   - `Sublime.app → File › Save with Encoding › UTF-8`

   > **NOTE**:
   >
   > Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return - linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).
   >
   > Make sure your PHP code files are also UTF-8 encoded to avoid collisions when concatenating strings with hardcoded or configured string constants.

2. All PHP files MUST end with a single blank line.

   - `Sublime.app → View › Line Endings › Unix`

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

5.2. FILENAME

Using lower-case letters, file names SHOULD be descriptive. Hyphens SHOULD separate words.

1. **Letters** MUST be all lower-case.
   - e.g., `autoloader.php`
2. Words MUST be separated with a hyphen.
   - e.g., `dms-config.php`

+ Example:

```
1.  divtel-suncom-config-update.php
2.
```

Figure 1: Filenames

▲

6. PHP Skeleton

A basic PHP file and its minimum requirements are provided in the following section.

Line-by-Line breakdown:

- Line 1: PHP open tag
- Line 2: Blank line
- Line 3: Your code
- Line 4: Blank line
- Line 5: End-of-file comment
- Line 6: Blank line

+ Example:

```
1.<?php
2.
3.// divtel code
4.
5.// EOF
6.
```

Figure 2: PHP Skeleton

▲

7. PHP TAGS

In the following section, PHP tags are addressed in how they are used in both PHP and PHP/HTML files.

The closing tag `?>` should not be included for files containing only PHP code. PHP does not require it. This will prevent trailing white spaces from being accidentally injected into a session, causing errors.

1. Open Tag MUST be on its own line and MUST be followed by a blank line.

> **NOTE**:
>
> PHP code should always be delimited by the full PHP code `<?php ?>`, not by the `<? ?>` shorthand. This is the most portable way of including PHP code on different operating systems and setups.

2. Close Tag MUST NOT be used in PHP code.
   - i.e., no `?>`
3. Open/Close Tag MUST be on a single line in PHP/HTML files.
   - i.e., `<?php . ?>`

> **NOTE**:
>
> When embedding multi-line PHP snippets within an HTML block, the PHP open and close tags must be on a line by themselves.

4. Short Open Tag MUST be used.
   - i.e., `<?` → `<?php`
5. Short Echo Tag SHOULD be used in PHP/HTML files.
   - i.e., `<?php echo` → `<?=`

> **NOTE**:
>
> To maximize compatibility, it is recommended to only use normal tags `<?php ?>` and `<? = ?>` Using short tags should be avoided when developing applications or libraries that are meant for redistribution, or deployment on PHP servers which are not under your control, because short tags may not be supported on the target server. For portable, redistributable code, be sure not to use short tags.
>
> Also, they can pose a security risk, and create conflict with processing instructions like `<?xml ... ?>`.

▲

7.1. OPEN TAG

An **Open** tag MUST be placed on its own line and MUST be followed by a blank line.

✗ **Incorrect**: because `<?php` is not on a separate line.

```
1.  <?php print_welcome_message();
2.
```

Figure 3: Open Tag – Incorrect

✗ **Incorrect**: because `<?php` MUST be followed by a blank line.

```
1.<?php
2.print_welcome_message();
3.
```

Figure 4: Open Tag – Incorrect

✓ **Correct**:

```
1.<?php
2.
3.print_welcome_message();
4.
```

Figure 5: Open Tag – Correct

▲ PHP Tags

7.2. CLOSE TAG

It is recommended that a closing PHP tag be omitted in a file that contains only PHP code so that occurrences of accidental whitespace or new lines that are added after the PHP closing tag can be avoided, which can trigger output buffering that causes uncalled effects.

In PHP files, the **Close** Tag MUST NOT be used.

✗ **Incorrect**: because `?>` was used.

```
1.  <?php
2.
3.  print_welcome_message();
4.
5.  ?>
6.
```

Figure 6: Close Tag – Incorrect

✓ **Correct**

```
1.  <?php.5
2.
3.  print_welcome_message();
4.
```

*Figure 7: Close Tag – **Correct***

▲ PHP Tags

7.3. OPEN/CLOSE TAG

PHP/HTML files MUST have the **Open/close** tags on its own single line.

✗ **Incorrect**: since `<?php` and `?>` are not on the same line.

```
1.<div>
1.<h1> <?php
2.              print_welcome_message();
3.?></h1>
2.</div>
3.
```

Figure 8: Open/Close Tag – Incorrect

✓ **Correct**

```
1.  <div>
2.      <h1> <?php print_welcome_message(); ?></h1>
3.  </div>
4.
```

Figure 9: Open/Close Tag – Correct

▲ PHP Tags

7.4. SHORT OPEN TAG

The short tags begin with `<?` and end with `?>`. Short style tags are only available when they are enabled in the php.ini configuration file on servers.

The **Short open** Tag MUST NOT be used.

✗ **Incorrect**: due to the use of `<?` instead of `<?php`.

```
1.  <?
2.
3.  print_welcome_message();
4.
```

Figure 10: Short Open Tag – Incorrect

✓ **Correct**

```
1.  <?php
2.
3.  print_welcome_message();
4.
```

Figure 11: Short Open Tag –Correct

▲ PHP Tags

7.5. SHORT ECHO TAG

What about `<?=`?

Although `<?` causes conflicts with XML, `<?=` does not. The options for turning it on and off were tied to the short_open_tag. Several issues were associated with short_open_tag that outweighed the benefits of `short_echo`, and as a result, you will find a million and a half recommendations to turn off the `short_open_tag`.

Since an earlier version of PHP, the short echo tag rather than `<?php print() ?>` has been re-enabled independently of the short_open_tag option. This clearly endorses the convenience of `<?=`, Since it isn't fundamentally flawed in and of itself.

PHP/HTML files SHOULD use the **Short echo** tag.

✓ **Acceptable**: `<?=` but should be used over `<?php echo` when possible.

```
1. <div>
2.                 <p> <?php echo get_welcome_message(); ?></p>
3. </div>
4.
```

Figure 12: Short Echo Tag – Acceptable

✓ **Preferred**:

```
1.      <div>
2.                 <p> <?= get_welcome_message(); ?></p>
3.      </div>
4.
```

Figure 13: Short Echo Tag – Preferred

Best Practices Short Open Tag.

▲ PHP Tags

7.6. NO SHORTHAND PHP

Important: Shorthand PHP **Start** tags SHOULD never be used. Always use **full** PHP tags.

✗ **Incorrect**:

```
1.  <? ... ?>
2.  <?= $var ?>
3.
```

Figure 14: No Shorthand PHP – Incorrect

✓ **Correct**:

```
1.  <?php ... ?>
2.  <?php echo $var; ?>
3.
```

Figure 15: No Shorthand PHP – Correct

▲ PHP Tags

7.7. End of File

The end of every PHP file is explained in this section.

End-of-file comment:

- MUST be placed at the end of each file
  - i.e., `// EOF`
- Must be placed on its own line
  - i.e., `// EOF`
- MUST have blank lines surrounding `//EOF`
  - i.e., `// EOF`

❌ **Incorrect**: because `// EOF` is missing.

```
1.  <?php
2.
3.  print_welcome_message();
4.
```

Figure 16: End of File – Incorrect

❌ **Incorrect**: because `// EOF` is not on a separate line.

```
1.  <?php
2.
3.  print_welcome_message(); // EOF
4.
```

Figure 17: End of File – Incorrect

❌ **Incorrect**: because there are no blank lines surrounding `// EOF`.

```
1.  <?php
2.
3.  print_welcome_message();
4.  // EOF
5.
```

Figure 18: End of File – Incorrect

✓ **Correct:**

```
<?php

print_welcome_message();

// EOF

```

Figure 19: End of File – Correct

▲ PHP Tags

8. Naming Conventions

In the art of software development, naming is consistently undervalued. Everyone seems to agree that having a nice name is a nice thing. In the end, many developers use cryptic abbreviations (to save typing). Spending 15 minutes (or more) just to name a method is well worth the effort! Many reasons exist for using proper names, and they all lead to better readable, manageable, stable, and secure code in the end.

Note that all class names, method names, comments, variables names, database table names, and field names must use English words (or abbreviations as necessary).

Make sure that you stick to a particular naming convention. Regardless of whether it is camelCase, PascalCase, snake_case, or any other case, it must remain consistent.

S-I-D

A name must be short, intuitive and descriptive:

- **Short**: A name must not take long to type and, therefore, remember;
- **Intuitive**: A name must read naturally, as close to the common speech as possible;
- **Descriptive**: A name must reflect what it does/possesses in the most efficient way.

▲

8.1. CONSTANT NAMES

In general, constants SHOULD be all upper-case and separated by underscores. Underscores can separate words - you can also use underscores to thematically group constants.

Parameters:

- **name**: The name of the constant
- **value**: The value of the constant - boolean, integer, float, string, or array. (You will learn about these in the data types chapter)
- case-insensitive: Specify whether the constant is case-insensitive or not. The default is false (case-sensitive)

✗ **Incorrect**: because it is not all upper-case, welcome_Message, Welcome_Message, and welcome_message.

```php
<?php

define('welcome_Message', '');
define('Welcome_Message', '');
define('welcome_message', '');

// EOF

```

Figure 20: Constant Names – Incorrect

✗ **Incorrect**: because there is no underscore between `WELCOME` and `MESSAGE`.

```php
<?php

define('WELCOMEMESSAGE', '');

// EOF

```

Figure 21: Constant Names – Incorrect

✓ **Correct**:

```php
<?php

define('WELCOME_MESSAGE', '');

// EOF

```

Figure 22: Constant Names – Correct

> **NOTE**:
>
> The `true`, `false` and `null` constants are excepted from the all-uppercase rule, and MUSTalways be lowercase.

See Also: Constants

▲ Naming Conventions

8.2. FUNCTION NAMES

Function naming is almost the same as variable naming except for the $ sign at the beginning. Functions do not have the `$` sign.

- A function name MUST start with a letter or underscore
- A function name MUST NOT start with a number
- Letters, numbers, and underscores can be used after the first letter in a function
- A function name is case-insensitive. Both `suncom()` and `SUNCOM()` refer to the same function

> **NOTE**:
>
> The `true`, `false` and `null` constants are excepted from the all-uppercase rule, and MUST always be lowercase.

-

> **TIP**:
>
> Always name functions with a name that describes the usage of the function.
> See Also: Function Prefix

✓ **Correct:**

```
1.  function suncomProjects ($name) {
2.          echo 'Currents projects: ' . $name;
3.          echo '<br>'; // line break
4.  }
5.          suncomProject (CRM Admin');
6.          suncomProject (CRM Admin (Beta)');
7.          suncomProject (CRM (Beta 2)');
8.
9.// EOF
10.
```

Figure 23: Function Names– Correct

See Also: Function Name

See Also: Function Prefix

See Also: Functions inside Loops

▲ Naming Conventions

8.3. VARIABLE NAMES

Variable, action/filter, and function names SHOULD always be lower-case (never Camel-Case). Variable names are case-sensitive. Use underscores to separate words. Variable names shouldn't be abbreviated unnecessarily; code SHOULD be unambiguous and self-documenting.

1. The `$` identifier should be added in front of the variable's name.
2. A variable name MUST start with a letter or underscore.
3. A variable name CANNOT start with a number.
4. A variable name CANNOT be `$this.` ($this is a special variable that cannot be assigned. After the first character, a variable name can contain letters, numbers, and characters between 127-255 ASCII.

✗ **Incorrect:** the naming of variables:

```
1.  $sObjRgstry
2.
3.  $argArr
4.
5.  $cx
6.
```

Figure 24: Variable Names – Incorrect

✓ **Correct:**

```
1.function suncom_users( $my_variable ) { [...] }
2.
```

Figure 25: Variable Names– Correct

✓ **Correct:**

```
1.  $singletonObjectsRegistry
2.
3.   $argumentsArray
4.
5.   $aLotOfHtmlCode
6.
```

Figure 26: Variable Names– Correct

See Also: Variables

▲ Naming Conventions

8.4. Class Names

Classes SHOULD be named descriptively. Abbreviations SHOULD be avoided whenever possible. Capitalized words separated by underscores (Camel_Case) SHOULD be used for class names. All acronyms SHOULD be capitalized.

+ Example:

```
1.  class Seminole_Category extends seminole { [...] }
2.  class DMS_HTTP { [...] }
3.
```

Figure 27: Class Names

Names for class files SHOULD be based on the class name with 'class'- prefixed and underscores in the class name replaced with hyphens.

+ Example:

```
1.  class-seminole-addin.php
2.
```

Figure 28: Class Names

✕ **Incorrect:** because `DivTelProgram` does not begin with a capital letter.

```
1.  <?php
2.
3.  namespace MyFlorida\DivTel;
4.
5.  class divtelProgram
6.  {
7.          // ...
8.  }
9.
10. // EOF
11.
```

Figure 29: Class Names – Incorrect

✗ **Incorrect:** because `DivTelProgram` is not in Camel-Case.

```
1.  <?php
2.
3.  namespace MyFlorida\DivTel;
4.
5.  class DivTelprogram
6.{
7.     // ...
8.}
9.
10. // EOF
11.
```

Figure 30: Class Names – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  namespace MyFlorida\DivTel;
4.
5.  class ACCESSFloridaSystem
6.  {
7.          // ...
8.  }
9.
10. // EOF
11.
```

Figure 31: Class Names – Incorrect

See Also: Class Name

▲ Naming Conventions

8.5. METHOD NAMES

The names of all methods are written in lower camelCase. For method names, only the characters a-z, A-Z, and 0-9 are allowed in order to prevent problems with different file systems. DO NOT use special characters.

Keep method names concise and descriptive at the same time.

Make method names descriptive, but keep them concise at the same time. Constructors MUST always called. `__construct()`, class names SHOULD never be used as method names.

- myMethod()
- someNiceMethodName()
- betterWriteLongMethodNamesThanNamesNobodyUnderstands()
- singYmcaLoudly()
- __construct()

See Also: Class Methods

▲ Naming Conventions

9. NAMESPACES

Namespaces are qualifiers that solve two different problems:

As an example, you might have classes that describe HTML tables, such as Table, Row, and Cell, and another for describing furniture, such as Chair, Table, and Bed. Namespaces can be used to organize classes into two different groups and prevent classes like Table and Table from being confused.

- They allow for a better organization by grouping classes that work together to perform a task
- They allow the same name to be used for more than one class

The use of one or more namespaces is described in this section, along with the naming convention. Namespaces are primarily intended for categorization and ordering.

1. Namespaces can only contain the characters `a-z`, `A-Z`, and `0-9`.
2. Namespace Declaration MUST appear as the first statement, and a blank line MUST follow it.
   - i.e., `<?php ↵ ↵ namespace MyFlorida; ↵ ↵ ..`
3. Namespace Name MUST begin with a capital letter and be written in upper Camel-Case.
   - e.g., `namespace MyFlorida;`
4. Multiple Namespaces MUST be used with the curly brace syntax.
   - i.e., `namespace MyFlorida { ... }`

> **NOTE**:
>
> Nested namespaces are not allowed. See: **Error! Reference source not found.**

5. Magic Constant SHOULD be used in order to reference the namespace name.
   - i.e., `__NAMESPACE__`

TIP:

In order to write future-proof code, it is recommended that you don't place many variables, functions or classes in the global namespace. This will prevent naming conflicts with 3rd party code as well as possible future additions to the language.

▲

9.1. NAMESPACE DECLARATION

The following sections describe how to use one or more namespaces and their naming convention.

The namespace declaration MUST be placed as the first statement, followed by a blank line.

✗ **Incorrect**: since the namespace, `MyFlorida;` isn't the first statement.

```
1.  <?php
2.
3.  print_welcome_message();
4.
5.  Namespace MyFlorida;
6.
7.  // EOF
8.
```

Figure 32: Namespace Declaration – Incorrect

✗ **Incorrect**: because there is no blank line following the namespace `MyFlorida`.

```
1.  <?php
2.
3.  Namespace MyFlorida;
4.  print_welcome_message();
5.
6.  // EOF
7.
```

Figure 33: Namespace Declaration – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  Namespace MyFlorida;
4.
5.  print_welcome_message();
6.
7.  // EOF
8.
```

Figure 34: Namespace Declaration – Correct

▲ Namespaces

9.2. NAMESPACE NAME

The first statement the PHP interpreter should encounter in a PHP file is the namespace definition. Only a declaration statement can appear above a namespace declaration, which is only if the script's encoding is being declared.

The namespace keyword is used to declare namespaces. Namespace names should follow the same PHP rules as other identifiers. So, the namespace must begin with a letter or underscore, followed by any number of letters, numbers, or underscores.

Namespace names MUST begin with a capital letter and be written in Camel-Case.

✕ **Incorrect**: because `myFlorida` does not begin with a capital letter.

```
1.  <?php
2.
3.  Namespace myFlorida;
4.
5.      // EOF
6.
```

Figure 35: Namespace Name – Incorrect

✕ **Incorrect**: since `MyFLORIDA` is not written in Camel-Case.

```
1.  <?php
2.
3.  Namespace MyFLORIDA;
4.
5.  // EOF
6.
```

Figure 36: Namespace Name – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  Namespace MyFlorida;
4.
5.  // EOF
6.
```

Figure 37: Namespace Name – Correct

▲ Namespaces

9.3. MULTIPLE NAMESPACES

Multiple namespaces may also be declared in the same file.

The curly-brace syntax MUST be used for multiple namespaces.

Example: Declaring multiple namespaces.

```
1.<?php
2.namespace DivTelProject;
3.
4.const CONNECT_OK = 1;
5.class Connection { /* ... */ }
6.function connect() { /* ... */  }
7.
8.namespace MyFlorida;
9.
10. const CONNECT_OK = 1;
11. class Connection { /* ... */ }
12. function connect() { /* ... */  }
13. ?>
14. // EOF
15.
```

Figure 38: Declaring Multiple Namespaces – Simple Combination Syntax

Example:

```
1.<?php
2.namespace DivTelProject {
3.
4.const CONNECT_OK = 1;
5.class Connection { /* ... */ }
6.function connect() { /* ... */  }
7.}
8.
9.namespace MyFlorida {
10.
11. const CONNECT_OK = 1;
12. class Connection { /* ... */ }
13. function connect() { /* ... */  }
14. }
15.
16. // EOF
17.
```

Figure 39: Declaring multiple namespaces – bracketed syntax

❌ **Incorrect**: this is due to the presence of two namespaces and the lack of curly braces.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  Namespace MyFlorida\View;
6.
7.  // EOF
8.
```

Figure 40: Multiple Namespaces – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel {
4.          // DivTel body
5.  }
6.
7.  Namespace MyFlorida\View {
8.          // View body
9.  }
10.
11. // EOF
12.
```

Figure 41: Multiple Namespaces – Correct

PHP does not allow nesting namespaces.

❌ **Incorrect**:

```
1.  <?php
2.  namespace MyFlorida\DivTel; {
3.  namespace nested {
4.      class DivTelProgram {}
5.  }
6.  }
7.
8. // EOF
9.
```

Figure 42: Multiple Namespaces: Not Allowed

However, nested namespaces are easy to simulate as follows:

✓ **Correct**:

```php
<?php

namespace MyFlorida\DivTel\Nested {
    class foo {}
}

// EOF

```

Figure 43: Multiple Namespaces: Nested

▲ Namespaces

9.4. MAGIC CONSTANT

The magic constants in PHP are predefined constants that change based on their use. They begin with a double underscore __ and end with a double underscore. These constants are similar to other predefined constants, but they are called magic constants since their values change with context.

To reference a namespace name, a magic constant SHOULD be used.

✓ **Acceptable:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel {
4.  // licensing body
5.  }
6.
7.  Namespace MyFlorida\DivTel {
8.  $welcome_message = MyFlorida\DivTel\get_welcome_message();
9.  }
10.
11. // EOF
12.
```

Figure 44: Magic Constant – Acceptable

✓ **Preferred:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel {
4.    // ModuleOne body
5.  }
6.
7.  Namespace MyFlorida\DivTel {
8.    $welcome_message  = __NAMESPACE__  . '\\' . get_welcome_message();
9.  }
10.
11. // EOF
12.
```

Figure 45: Magic Constant – Preferred

See Also: Magic constants **Error! Reference source not found.Error! Reference source not found.**

▲ Namespaces

10. Comments

The following sections describe the format and use of comments.

In general, if you see a section of code and think, "Wow, I don't want to try and explain that," you should comment on it before you forget how it works. In addition, if you are working with a group or plan to have anyone else use your script, the comments will inform the other programmers of what you did at each step, making it much easier for them to work with and edit your code if needed.

PHP drew its inspiration from several programming languages, most notably C, Perl, and Unix shell scripts. As a result, PHP supports styles of comments from all those languages, and those styles can be intermixed freely in PHP code. However, the following rules are intended to provide guidance for commenting on PHP code in a standardized manner:

- Rule 1: Comments should not duplicate the code
- Rule 2: Good comments do not excuse unclear code
- Rule 3: If you can't write a clear comment, there may be a problem with the code
- Rule 4: Comments should dispel confusion, not cause it
- Rule 5: Explain unidiomatic code in comments
- Rule 6: Provide links to the original source of copied code
- Rule 7: Include links to external references where they will be most helpful
- Rule 8: Add comments when fixing bugs
- Rule 9: Use comments to mark incomplete implementations

The following is a list of what should be documented in PHP files:

- Functions and class methods
- Classes
- Class members (including properties and constants)
- Requires and includes
- Hooks (actions and filters)
- Inline comments
- File headers
- Constants

1. Single-line Comments MUST be formatted with two forward slashes.
   - e.g., `// My comment`
2. Multi-line Comments MUST be presented in a block format.
   - i.e., `/** ↲ * My comment ↲ */`
3. Header Comments SHOULD be formatted in a block format.
   - i.e., `/** ↲ * Name of code section ↲ */`
4. Divider Comments SHOULD be formatted using the block format with asterisks between each comment.
   - i.e., `/** 75 asterisks */`
5. Comments MUST be placed on their own line.
   - i.e., `↲ // My comment`
6. Blocks of Code SHOULD always be explained or summarized.
   - e.g., `// Compare user accounts from export against expired accounts in system`
7. Ambiguous Numbers MUST be clarified.
   - e.g., `// 1,000 processable records per hour API limit`
8. External Variables MUST be clarified.
   - e.g., `// Database object included in file.php`

▲

10.1. Single-line Comments

The single-line comment tells the interpreter to ignore everything that occurs on that line to the right of the comment. To do a single line comment, type `//`, and all text to the right will be ignored by the PHP interpreter.

Each line of a comment MUST begin with two forward slashes.

✗ **Incorrect**: the single-line comment is formatted with `/*` and `*/`.

```
1.  <?php
2.
3.  /* This is a comment */
4.
5.  // EOF
6.
```

Figure 46: Single-Line Comment – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  // This is a comment
4.
5.  // EOF
6.
```

Figure 47: Single-Line Comment – Correct

▲ Comments

10.2. Multi-line Comments

The PHP comment can be used to comment out large code blocks or write multiple-line comments. The multiple-line PHP comment begins with `/*` and ends with `*/`.

Comments with more than one line MUST be structured as a block.

✗ **Incorrect**: because `//` is used for a multi-line comment.

```
1. <?php
2.
3. // This is a
4. // multi-line
5. // comment
6.
7. // EOF
8.
```

Figure 48: Multi-Line Comment – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  /**
4.  * This is a
5.  * multi-line
6.  * comment
7.  */
8.
9.  // EOF
10.
```

Figure 49: Multi-Line Comment – Correct

▲ Comments

10.3. Header Comments

There must be a "page-level" doc-block at the top of each file and a "class-level" doc-block immediately above each class.

Comments in the header SHOULD be structured in block format.

+ Example:

```
1.    <?php
2.
3.    /**
4.     * @category    CategoryName
5.     * @package     PackageName
6.     * @author      Original Author
7.     * @author      Another Author
8.     * @copyright   1997-2022 The PHP Group
9.     * @license     http://www.php.net/license/3_01.txt  PHP License 3.01
10.  * @version     SVN: $Id$
11.  * @link        http://pear.php.net/package/PackageName
12.  * @see         NetOther, Net_Sample::Net_Sample()
13.  * @since       File available since Release 1.2.0
14.  * @deprecated  File deprecated in Release 2.0.0
15.
16.  /**
17.  * Global CSAB settings
18.  */
19.
20.   define('CSAB_ONE', '');
21.   define('CSAB_TWO', '');
22.   Define('CSAB_THREE', '');
23.
24.  // EOF
25.
```

Figure 50: Comments – Header Comments

See Also: Class Documentation

▲ Comments

10.4. Divider Comments

Divider comments SHOULD be written in block format with 75 asterisks between them.

✗ **Incorrect**: this is incorrect since `#` is used instead of `*`.

```
1.  <?php
2.
3.  /**###########################################################################*/
4.
5.  // EOF
6.
```

Figure 51: Divider Comments – Incorrect

✗ **Incorrect**: because there SHOULD be a 75 `*` instead of 10.

```
1.  <?php
2.
3.  /**********/
4.
5.  // EOF
6.
```

Figure 52: Divider Comments – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  /**
4.  * Beginning + Middle + End
5.  * 3 spaces + 75 spaces + 2 spaces = 80 character line limit
6.  */
7.
8.  /*****************************************************************************/
9.
10. // EOF
11.
```

Figure 53: Divider Comments – Correct

▲ Comments

10.5. COMMENTS

- Documentation and comments SHOULD be formatted properly, with proper grammar and punctuation
- Insert a space between the comment character `*` or `//` and the first letter of the sentence
- Document the current version of the code, not the differences from past versions or future plans (except for `@todo` and `@deprecated` tags)
- Comments and variable names should be in English, using US English spelling (e.g., "color" rather than "colour")
- If referring to a module or theme, use words and capitalization like `"the Foo Bar module"` (in other words, the module/theme name is a proper noun; do not include the words "module" or "theme" in the proper noun)
- With a few exceptions, lines containing comments (including doc-blocks) MUST wrap as close to 80 characters as possible
- Two spaces indent documentation following tags such as `@param`, `@return`, etc.
- Class members (functions, properties, constants), interfaces, interface functions, classes, and files must all be documented; even private class members
- Summary lines (first lines of doc-blocks) must be under 80 characters, begin with a capital letter, and end with a period `.`. They MUST describe briefly what a function does, what a class does, what a file contains, etc.
- Each comment MUST be on its own line

✗ **Incorrect:** since `//` prints welcome message is not on its own line.

```
1.  <?php
2.
3.  print_welcome_message(); // Prints welcome message
4.
5.  // EOF
6.
```

Figure 54: Comment – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  // Prints welcome message
4.  print_welcome_message();
5.
6.  // EOF
7.
```

Figure 55: Comment – Correct

▲ Comments

10.6. Blocks of Code

Comments are usually written within the block of PHP code to explain or summarize the code's functionality.

Comment blocks are typically written within functions, between 'chunks' of code. Also, they can document other parts of code, for example, at the start of data chunks.

Depending on the size of the block comment, there are three classifications:

- A single line comment usually describes a simple item

  See Also: Single-line Comments

- Several lines of comments, up to about five lines, can summarize a longer and more complex item or set of items

  See Also: Multi-line Comments

- A longer comment will describe even more, although this is now tending towards a major header comment

  See Also: Header Comments

In general, a block comment has a cost in vertical space and should offer value for money. For instance, having a block comment larger than the code it describes may be overkill.

Usually, a block comment describes the code below it. This association can be made clearer by using some method to explicitly associate it more closely with the code that it describes.

Using blank lines:

A simple principle is to use a blank line above the comment to physically place it closer to the line below than the line above:

✓ **Acceptable:** but the code block SHOULD be explained or summarized.

```php
<?php

foreach ($users as $user) {
      if ($expr1) {
        // ...
} else {
        // ...
}
    if ($expr2) {
                  // ...
        } elseif ($expr3) {
                  // ...
        } else {
                  // ...
        }
                  // ...
}

// EOF

```

Figure 56: Blocks of Code – Acceptable

✓ **Preferred:**

```php
<?php

/**
* Get active DMS user's updated resume with profile photo for CSAB update.
* If no resume exists, check human resources.
* If neither has a resume, send the user email to upload one.
*/
foreach ($users as $users) {
        if ($expr1) {
                  // ...
  } else {
                  // ...
  }
  if ($expr2) {
                  // ...
  } elseif ($expr3) {
                  // ...
  } else {
                  // ...
  }
  // ...
}

// EOF

```

Figure 57: Blocks of Code – Preferred

▲ Comments

10.7. Ambiguous Numbers

Numbers that may be ambiguous MUST be clarified.

✗ **Incorrect:** `1000` is incorrect since it has not been clarified.

```
1.  <?php
2.
3.  while ($expr && $x < 1000) {
4.          // ...
5.  }
6.
7.  // EOF
8.
```

Figure 58: Ambiguous numbers – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  // Script times out after 1,000 records
4.  while ($expr && $x < 1000) {
9.          // ...
5.  }
6.
7.  // EOF
8.
```

Figure 59: Ambiguous numbers – Correct

▲ Comments

10.8. External Variables

There MUST be clarity with external variables.

✗ **Incorrect:** since the source of `$users` is unclear.

```php
<?php

include_once 'CSAB-file.php';

// ...

foreach($users as $user) {
        // ...
}

// EOF

```

Figure 60: External Variables – Incorrect

✓ **Correct:**

```php
<?php

include_once 'CSAB-file.php';

// ...

// $users from CSAB-file.php
foreach($users as $user) {
        // ...
}

// EOF

```

Figure 61: External Variables – Correct

▲ Comments

11. INCLUDES

Include and required files are described in this section, along with the requirements for using the function.

Use `require_once` whenever you unconditionally include a file use, `include_once` Whenever you conditionally include a file. In either case, files will only be included once. As they share the same file list, you don't need to worry about mixing them. Files included by `require_once` will not be included by `include_once`.

NOTE:

`Include_once` and `require_once` are PHP language statements, not functions. The correct formatting is:

```
1.  require_once JPATH_COMPONENT .'/helpers/helper.php';
```

You should not enclose the filename in parentheses.

1. Include/Require Once SHOULD be used.
   - i.e., include→ `include_once`, require→ `require_once`
2. Parenthesis MUST NOT be used.
   - e.g., `Include_once(file.php');` → `include_once 'file.php';`
3. Purpose of Include MUST be documented by providing a comment.
   - e.g., `// Provides DMS environment require_once 'csab-load.php';`

▲

11.1. INCLUDE/REQUIRE ONCE

In the `include_once` statement, the specified file is included and evaluated during the script execution. The behavior is similar to the include statement. However, if the code from a file has already been included, it will not be included again, and `include_once` returns true. As the name suggests, the file will be included just once.

Include/require once SHOULD be used.

✓ **Acceptable:** if possible, `_once` SHOULD be appended to `include` and `require`.

```
1.  <?php
2.
3.  include 'csab-file.php';
4.  require 'another-csab-file.php';
5.
6.  // EOF
7.
```

Figure 62: Include/Require Once – Acceptable

✓ **Preferred:**

```
1.  <?php
2.
3.  include_once 'some-csab.php';
4.  require_once 'another-csab-file.php';
5.
6.  // EOF
7.
```

Figure 63: Include/Require Once – Preferred

▲ Includes

11.2. PARENTHESIS

There MUST NOT be the use of parenthesis, as it would be redundant.

✗ **Incorrect:** Include_once and require_once are used within parenthesis.

```
1.  <?php
2.
3.  include_once('csab.php');
4.  require_once('another-csab-file.php');
5.
6.  // EOF
7.
```

Figure 64: Parenthesis – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  include_once 'csab.php';
4.  require_once 'another-csab-file.php';
5.
6.  // EOF
7.
```

Figure 65: Parenthesis – Correct

▲ Includes

11.3. PURPOSE OF INCLUDE

Include (or require) statements; take all text/code/markup contained in a specified file and copy it into the PHP file that uses the statement. Including files is very useful when you want to include the same PHP, HTML, or text on multiple website pages.

An explanation of the purpose of an include MUST be provided.

✗ **Incorrect**: there is no explanation for what `csab-file.php` does or provides.

```
1.  <?php
2.
3.  require_once 'csab-file.php';
4.
5.  // EOF
6.
```

Figure 66: Purpose of Include – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  // Provides oranization framework
4.  require_once 'csab-file.php';
5.
6.  // EOF
7.
```

Figure 67: Purpose of Include – Correct

▲ Includes

12. Code Format

Consistently using the same style throughout the code makes it easier to read. Well-written code is easier for you to understand and for other collaborators to comprehend and appreciate. Adhering to properly formatted code reduces errors and makes collaborating on software easier.

The rules in this section pertain to the use of whitespace, text, and code formatting, in general.

4. Case Sensitivity:
    - PHP is partially case-sensitive:
        - **Case Sensitive**
            - Strings
            - Variables
            - Array keys
            - Object Properties
            - Constants, by default
        - **Case Insensitive**
            - Key Words (if, else, null, foreach, echo, etc.)
            - Functions
            - Object Methods
            - Constants, if defined accordingly
            - Class Name

NOTE:

Classes are thus a mixed bag:

- The class keyword is case insensitive
- Class names are case insensitive, for declaration, instantiation, and static calls
- Class methods, being functions, are case insensitive
- Class properties, being variables & constants, are case sensitive

Because Strings are case sensitive, anything that relies on strings, such as array keys and values, is also case sensitive.

5. Line Length MUST not exceed 80 characters unless the line is text.
   - i.e., `|---- 80+ chars ----|` → refactor expression and/or break list values
6. Line Indentation MUST always be accomplished by using tabs.

   See Also: Line Indentation **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**
7. Blank Lines SHOULD be inserted between logical blocks of code.
8. Text Alignment MUST only be accomplished with the use of spaces.
   - i.e., `$var . . . = .;`

12.1. TRAILING WHITESPACE

Remove trailing whitespace at the end of each line of code. Omitting the closing PHP tag at the end of a file is preferred. There MUST be no trailing whitespace after statements or serial comma break or on blank lines.

✗ **Incorrect**: because there are two spaces after `$quotes_exist = false;`

```
1.  <?php
2.
3.  $quotes_exist = false;..
4.
5.  print_welcome_message();
6.
7.  // EOF
8.
```

Figure 86: Trailing Whitespace – Incorrect

✗ **Incorrect**: because there is a space after `,.`

```
1.  <?php
2.
3.  $my_movies = array(
4.  'Gone with the Wind',.
5.  'The Wizard of Oz',.
6.  'The Lord of the Rings',.
7.  'E.T. the Extra-Terrestrial'
8.  );
9.
10. // EOF
11.
```

Figure 87: Trailing Whitespace – Incorrect

✗ **Incorrect**: because the blank line below `$quotes_exist = false;` has two spaces.

```
1.  <?php
2.
3.  $quotes_exist = false;
4...
5.  print_welcome_message();
6.
7.  // EOF
8.
```

Figure 88: Trailing Whitespace – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  $quotes_exist = false;
4.
5.  print_welcome_message();
6.
7.  // EOF
8.
```

Figure 89: Trailing Whitespace – Correct

✓ **Correct**:

```
1.  <?php
2.
3.  $my_movies = array(
4.          'Gone with the Wind',
5.          'The Wizard of Oz',
6.          'The Lord of the Rings',
7.          'E.T. the Extra-Terrestrial'
8.  );
9.
10. // EOF
11.
```

Figure 90: Trailing Whitespace – Correct

▲ Code Format

12.2. WHITESPACE INSENSITIVITY

Whitespace on the screen is usually not visible, including spaces, tabs, and end-of-line characters, such as carriage returns. This means many whitespace characters are the same as one whitespace character. A statement can essentially be broken up into several lines or a number of separate statements. Although whitespaces are generally ignored in PHP syntax, there are several places where you cannot have a space without affecting the sense (and the result). For instance, with a space in the middle of an operator, then it is no longer an operator. It is also true for numbers or in the middle of keywords, variables, constants, operators, and strings. Also, the opening parenthesis of the argument list or before the closing parenthesis. There MUST NOT be a space before each comma, and there MUST be one space after each comma.

In the following example spaces and tabs are used in a numeric operation, but in both cases, $four returns the same value.

Example:

```
1.  $four = 2 + 2; // single spaces
2.  $four <tab> = <tab2< tab> + <tab> 2 ; // spaces and tabs
3.  $four =
4.  2+
5.  2; // multiple lines
6.
```

Figure 91: Whitespace – Insensitivit – Example

The following two examples are the same:

Example: Advance whitespace insensitivity

```
1.<?php
2.function student_info
3.
4.$resource_name,
5.$role,
6.$company_name
7.)
8.{
9.      echo 'The Name of resource is: $resource_name <br />';
10.     echo 'His role is: $role and his project is with: $company_name';
11. }
12. student_info(
13. 'Stephen Mitchell', 'Technical Writer', 'CSAB for DivTel'
14. )
15. // EOF
16.
```

Figure 92: Advance Whitespace – Insensitivity

Output:

```
1.  The Name of the resource is: Stephen Mitchell
2.  His role is: Technical Writer and his project is with: CSAB for DivTel
3.
```

Figure 93 Advance Whitespace – Insensitivity – Output

Example:

```
1.<?php
2.  // single space between $xyz, =, 11, +, 12
3.  $xyz = 11 + 12;
4.  echo $xyz .'<br />';
5.  // tabs and spaces
6.  $xyz =    11 12;
7.  echo $xyz;
8.
9.  // EOF
10.
```

Figure 94: Advance Whitespace – Insensitivity

Output:

```
1.  23
2.    23
```

Figure 95: Advance Whitespace – Output

9. MUST NOT appear after a statement, a serial comma break, or after blank lines.
   - i.e., no . , , ↲ , ↲

12.3. SPACE USEAGE

ALWAYS put spaces after commas, and on both sides of logical, comparison, string and assignment operators.

Example:

```
1.        x === 23
2.        foo && bar
3.        ! foo
4.        array( 1, 2, 3 )
5.        $baz . '-5'
6.        $term .= 'X'
7.
```

Figure 96: Space Usage – Logical, Comparison, String, and Assignment Operators –Example

Ensure there are spaces between the opening and closing parentheses of if, elseif, foreach, for, and switch blocks.

Example:

```
1.  foreach ( $foo as $bar ) { ...
2.
```

Figure 97: Space Usage – Spaces between the Opening and Closing Parentheses –Example

When defining a function, do it in the following manner:

Example:

```
1.  function my_function( $param1, func_param( $param2 ) );
2.
3.  Function my_other_function(); {  .
4.
```

Figure 98: Space – UsageDefining a Function –Example

When performing logical comparisons, do it in the following manner:

Example:

```
1.  if ( ! $foo ) { ...
2.
```

Figure 99: Space Usage – Logical Comparisons –Example

Always use the short form of type cast: (int) rather than (integer) and (bool) instead of (boolean). For float casts use (float).:

Example:

```
1.  foreach ( (array) $foo as $bar ) { ...
2.
3.  $foo = (bool) $bar;
4.
```

Figure 100: Space Usage – Short Form of Type Cast –Example

When referring to array items, only include a space around the index if it is a variable.

Example:

```
1.  $x = $foo['bar']; // correct
2.  $x = $foo[ 'bar' ]; // incorrect
3.
4.  $x = $foo[0]; // correct
5.  $x = $foo[ 0 ]; // incorrect
6.
7.  $x = $foo[ $bar ]; // correct
8.  $x = $foo[$bar]; // incorrect
9.
```

Figure 101: Space Usage – Array Items –Example

When a case statement is used in a switch block, there must be no space before the colon.

Example:

```
1.  switch ( $foo ) {
2.        case 'bar': // correct
3.        case 'ba' : // incorrect}
4.
```

Figure 102: Space Usage – Case Statement is Used in a Switch Block –Example

Similarly, there should be no space before the colon on return type declarations.

Example:

```
1.  function sum( $a, $b ): float {
2.
3.  return $a + $b;
4.
```

Figure 103: Space Usage – Return Type Declarations –Example

Unless otherwise specified, parentheses should have spaces inside of them.

Example:

```
1.  if ( $foo && ( $bar || $baz ) ) { ...
2.
3.  my_function( ( $x - 1 ) * 5, $y );
4.
```

Figure 104: Space Usage – Parentheses Should Have Spaces – Example

▲ Code Format

10. Keywords MUST be written in lower case.
    - e.g., `false`, `true`, `null`, etc.

12.4. SPACE USEAGE

ALWAYS put spaces after commas, and on both sides of logical, comparison, string and assignment operators.

Example:

```
5.        x === 23
6.        foo && bar
7.        ! foo
8.        array( 1, 2, 3 )
9.        $baz . '-5'
10.       $term .= 'X'
11.
```

Figure 96: Space Usage – Logical, Comparison, String, and Assignment Operators –Example

Ensure there are spaces between the opening and closing parentheses of if, elseif, foreach, for, and switch blocks.

Example:

```
12. foreach ( $foo as $bar ) { ...
13.
```

Figure 97: Space Usage – Spaces between the Opening and Closing Parentheses –Example

When defining a function, do it in the following manner:

Example:

```
14. function my_function( $param1, func_param( $param2 ) );
15.
16. Function my_other_function(); {  .
17.
```

Figure 98: Space – UsageDefining a Function –Example

When performing logical comparisons, do it in the following manner:

Example:

```
18. if ( ! $foo ) { ...
19.
```

Figure 99: Space Usage – Logical Comparisons –Example

Always use the short form of type cast: (int) rather than (integer) and (bool) instead of (boolean). For float casts use (float).:

Example:

```
20. foreach ( (array) $foo as $bar ) { ...
21.
22. $foo = (bool) $bar;
23.
```

Figure 100: Space Usage – Short Form of Type Cast –Example

When referring to array items, only include a space around the index if it is a variable.

Example:

```
24. $x = $foo['bar']; // correct
25. $x = $foo[ 'bar' ]; // incorrect
26.
27. $x = $foo[0]; // correct
28. $x = $foo[ 0 ]; // incorrect
29.
30. $x = $foo[ $bar ]; // correct
31. $x = $foo[$bar]; // incorrect
32.
```

Figure 101: Space Usage – Array Items –Example

When a case statement is used in a switch block, there must be no space before the colon.

Example:

```
33. switch ( $foo ) {
34.       case 'bar': // correct
35.       case 'ba' : // incorrect}
36.
```

Figure 102: Space Usage – Case Statement is Used in a Switch Block –Example

Similarly, there should be no space before the colon on return type declarations.

Example:

```
37. function sum( $a, $b ): float {
38.
39. return $a + $b;
40.
```

Figure 103: Space Usage – Return Type Declarations –Example

Unless otherwise specified, parentheses should have spaces inside of them.

Example:

```
41. if ( $foo && ( $bar || $baz ) ) { ...
42.
43. my_function( ( $x - 1 ) * 5, $y );
44.
```

Figure 104: Space Usage – Parentheses Should Have Spaces – Example

▲ Code Format

11. KeywordsMUST be written in lower case, and an underscore MUST separate all words.
    - e.g., `$welcome_message`

    See Also: Variable Initialization

NOTE:

```
1.<?php
2.$SUNCOM=1;
3.$suncom=2;
4.$SUNcom=3;
5.$sunCOM=4;
6.6.
7.echo 'SUNCOM is $SUNCOM, $suncom is $suncom, SUNcom is $SUNcom,
    SUNcom is $SUNcom, and sunCOM is $sunCOM';
8.  //EOF
9.
```

```
1.  Result = SUNCOM is 1, suncom is 2, SUNcom is 3, SUNcom is 3, and
    sunCOM is 4.
```

12. Global Variables MUST be declared on a separate line and MUST be indented after the first declaration.
    - e.g., `global $var1, ↵ ⇥ $var2;`

13. Constants MUST be in all capital letters and separated by underscores.
    - e.g., `WELCOME_MESSAGE`
14. Statements MUST each be on their own line, followed by a semicolon.
    - e.g., `welcome_message();`
15. Operators MUST have a space on both sides of the operator.
    - e.g., `$total = 15 + 7;, $var .= '';`
16. Unary Operators MUST be connected to their variables or integers.
    - e.g., `$index++, --$index`
17. Concatenation Period MUST be enclosed by a space.
    - e.g., `echo 'Read:' . $welcome_message;`
18. Single Quotes MUST always be used.
    - e.g., `echo 'Hello, World!';`
19. Double Quotes SHOULD be avaoided.
    - e.g., `echo 'Read: $welcome_message';` → `echo 'Read: ' . $welcome_message;`

> **NOTE**:
>
> When appropriate, use single and double quotes. If the string isn't being evaluated, use single quotes. As long as you alternate your quoting style, you should almost never have to escape quotes in a string.

▲

12.5. Case Sensitivity

Although some PHP constructs are case-insensitive, such as function names and class names, variables, array keys, class constants, and class properties are case-sensitive. Please refer to the following section for more information regarding case sensitivity.

All variables names are case sensitive, these include normal variables and superglobals such as `$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, `$_SESSION`, `$_GLOBALS` etc.

Example:

```php
<?php
$abc = 'abcd';
echo $abc; //Output 'abcd'
echo $aBc; //No output
echo $ABC; //No output

// EOF

```

Figure 68: Case Sensitivity – Variables – Example

Constants name by default is case sensitive, usually use UPPER CASE for constant name.

Example:

```php
<?php
define('ABC','Hello World');
echo ABC;   //Output Hello World
echo abc;   //Output abc

      // EOF

```

Figure 69: Case Sensitivity – Constants – Example

See Also: Constants

Configuration in php.ini

+ Example:

```
1.  file_uploads=1 cannot be written as File_Uploads = 1
```

Figure 70: Case Sensitivity – php.ini –Example

Function names, method names, or class names are case-insensitive, but it is recommended that you use the same name as you define it and as set out in this code standard.

Function names:

+ Example:

```
1.  <?php
2.  function show(){
3.      echo 'Hello World';
4.    }
5.  show(); //Output Hello World
6.  SHOW(); //Output Hello World
7.
8.  // EOF
```

Figure 71: Case Sensitivity – Function Name –Example

Method name and class names:

+ Example:

```
1.  <?php
2.   class cls
3.
4.      {
5.      static function func(){
6.      echo 'hello world';
7.      }
8.      Cls::FunC();  //Output hello world
9.
10. // EOF
11.
```

Figure 72: Case Sensitivity – Method Name and Class Name – Example

See Also: Function Name

See Also: Class Methods, Class Name

Magic constants are case insensitive:

Example: Include `__LINE__`, `__FILE__`, `__DIR__`, `__FUNCTION__`, `__CLASS__`, `__METHOD__`

```
1.        <?php
2.            echo __line__;  //Output 2
3.        echo __LINE__;  //Output 3
4.
5.        // EOF
6.
```

Figure 73: Case Sensitivity – Magic Constants – Example

Example: `NULL`, `TRUE`, `FALSE` are case insensitive

```
1.<?php
2.    $a = null;
3.    $b = NULL;
4.
5.    $c = true;
6.    $d = TRUE;
7.
8.    $e = false;
9.    $f = FALSE;
10.
11.   var_dump($a == $b); //Output boolean true
12.   var_dump($c == $d); //Output boolean true
13.   var_dump($e == $f); //Output boolean true
14.
15. // EOF
16.
```

Figure 74: Case Sensitivity – NULL,TRUE, FALSE– Example

NOTE:

`Null`, `True`, and `False` are case insensitive. However, this code standard specifies that they should be presented in lower case.

See Also: Magic Constant

> **NOTE**:
>
> `TRUE` , `FALSE`. and `NULL` are predefined as upper-case in PHP. You can also use the lowercase versions, if you prefer, as they are also predefined. In fact, the lowercase versions are more stable, because PHP does not allow you to redefine them; the uppercase ones may be redefined—something you should bear in mind if you import third-party code.

Type casting:

(int), (integer) Convert to integer

(bool) (boolean) convert to boolean

(float) (double) convert to floating-point

(string) convert to string

(array) convert to array

(object) convert to object

12.6. LINE LENGTH

1. 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 unless it is text.
2. There MUST NOT be any trailing whitespace at the end of lines.
3. Blank lines MAY be added to improve readability and indicate related code blocks except where explicitly forbidden.
4. There MUST NOT be more than one statement per line.

✗ **Incorrect**: because there are more than 80 characters in the expression, it SHOULD be refactored.

```
1.  <?php
2.
    if(in_array('Gone with the Wind', $movies) && in_array('The Wizard of Oz', $movies) &&
    in_array('The Lord of the Rings', $movies) && in_array('E.T. the Extra-Terrestrial',
    $movies)) {
3.        // if body
4.  }
5.
6.  // EOF
7.
```

Figure 75: Line Length – Incorrect

✗ **Incorrect**: because arguments exceed 80 characters and MUST be placed on their own line.

```
1.  <?php
2.
3.  $my_movies = array('Gone with the Wind', 'The Wizard of Oz', 'The Lord of the Rings',
    'E.T. the Extra-Terrestrial');
4.
5.  // EOF
6.
```

Figure 76: Line Length – Incorrect

✓ **Acceptable**: since the line length was exceeded by text, not code.

```
<?php

$text = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec posuere rutrum tincidunt. Duis lacinia laoreet diam, nec consectetur magna facilisis eget. Quisque elit mauris, auctor quis vulputate id, sagittis nec tortor. Praesent fringilla lorem eu massa convallis ultricies. Maecenas lacinia porta purus, sollicitudin condimentum felis eu ipsum.';

// EOF

```

Figure 77: Line Length – Acceptable

✓ **Correct**:

```
<?php

$my_movies = array(
'Gone with the Wind',
'The Wizard of Oz',
'The Lord of the Rings',
'E.T. the Extra-Terrestrial'
);

$has_all_movies = true;

foreach($my_movies as $my_movie) {
        if(!in_array($my_movie, $movies)) {
                $has_all_movies = false;
        }
}

if($has_all_movies) {
        // if body
}

$some_long_variable = get_something_else(
        'from_some_other_function',
        'another_long_argument'
);

// EOF

```

Figure 78: Line Length – Correct

▲ Code Format

12.7. LINE INDENTATION

Indentation should always reflect logical structure. Indentation of lines MUST be done with real tabs and not spaces, allowing the most flexibility across clients.

> **NOTE:**
>
> Although, the ***PSR-12: Coding Style Guide*** insists code MUST use an indent of 4 spaces and MUST NOT use tabs for indenting. At the end of the day, tabs versus spaces is truly a matter of preference, however the tab is still the character specifically designed for indentation, and using one tab character per indentation level instead of 2 or 4 spaces will use less disk space / memory / compiler resources and the like.
>
> The argument for 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.
>
> Throughout this document, indent and indentation is referred to as a tab.
>
> For another point of view, See: Tabs vs. Spaces — A scientific approach to an endless debate.

✗ **Incorrect**: because the `echo WELCOME_MESSAGE` is indented with spaces and not tabs.

```
1.  <?php
2.
3.  function print_welcome_message() {
4.  echo WELCOME_MESSAGE;
5.  }
6.
7.  // EOF
8.
```

Figure 79: Line Indentation: – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  function print_welcome_message() {
4.  echo WELCOME_MESSAGE;
5.  }
6.
7. // EOF
8.
```

Figure 80: Line Indentation: – Correct

▲ Code Format

12.8. BLANK LINES

1. All PHP files MUST only use the Unix LF (linefeed) line ending. All PHP files MUST end with a non-blank line, terminated with a single LF.

> **NOTE**:
>
> Ending the file with a blank line can serve as a warning to help detect a possibly defective, truncated file, on the assumption that file without a newline at the end is suspect for being truncated.
>
> Utilities that operate on files like the diff utility may not cope well with lines that don't end with a newline.

2. Blank lines improve readability by setting off sections of code that are logically related.
3. Two blank lines should always be used in the following circumstances:
   - Between sections of a source file
   - Between class and interface definitions
4. One blank line should always be used in the following circumstances:
   - Between methods
   - Between the local variables in a method and its first statement
   - Before a block or single-line comment
   - Between logical sections inside a method to improve readability
   - Between each logical block of code, blank lines SHOULD be added

✓ **Acceptable**: but it can be more difficult to scan code.

```php
<?php

$my_movies = array(
'Gone with the Wind',
'The Wizard of Oz',
'The Lord of the Rings',
'E.T. the Extra-Terrestrial'
);
$has_all_movies = true;
foreach($my_movies as $my_movie)
        {
                if(!in_array($my_movie, $movies)) {
                $has_all_movies = false;
        }
}
if($has_all_movies)
{
        // if body
}

// EOF

```

Figure 81: Blank Lines – Acceptable

✓ **Preferred**:

```php
<?php

$my_movies = array(
'Gone with the Wind',
'The Wizard of Oz',
'The Lord of the Rings',
'E.T. the Extra-Terrestrial'
);

$has_all_movies = true;

foreach($my_movies as $my_movie) {
        if(!in_array($my_movie, $movies)) {
                $has_all_movies = false;
        }
}

if($has_all_movies) {
        // if body
}

// EOF

```

Figure 82: Blank Lines – Preferred

▲ Code Format

12.9. TEXT ALIGNMENT

The alignment of text MUST be accomplished by inserting spaces instead of tabs.

✗ **Incorrect**: because vertical alignment SHOULD use tabs instead of spaces `=>`.

```
1.  <?php
2.
3.  $movie_quotes = array(
4.          'gone_with_the_wind'     => 'As God is my witness, I'll never be hungry again.',
5.          'the_wizard_of_oz'       => 'I've got a feeling we're not in Kansas anymore.',
6.          'the_lord_of_the_rings'  => 'Not all those who wander are lost.',
7.          'the_extra_terrestrial'  => 'You could be happy here, I could take care of you.'
8.  );
9.
10. // EOF
11.
```

Figure 83: Text Alignment – Incorrect

✗ **Incorrect**: because the array keys are indented with spaces instead of tabs.

```
1.  <?php
2.
3.  $movie_quotes = array(
12.                 'gone_with_the_wind'    => 'As God is my witness, I'll never be hungry
    again.',
13.         'the_wizard_of_oz'      => 'I've got a feeling we're not in Kansas anymore.',
14.         'the_lord_of_the_rings' => 'Not all those who wander are lost.',
15.         'the_extra_terrestrial' => 'You could be happy here, I could take care of you.'
4.  );
5.
6.// EOF
7.
```

Figure 84: Text Alignment – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  $movie_quotes = array(
16. 'gone_with_the_wind'    => 'As God is my witness, I'll never be hungry again.',
17. 'the_wizard_of_oz'      => 'I've got a feeling we're not in Kansas anymore.',
18. 'the_lord_of_the_rings' => 'Not all those who wander are lost.',
19. 'the_extra_terrestrial' => 'You could be happy here, I could take care of you.'
4.  );
5.
6.  // EOF
7.
```

Figure 85: Text Alignment – Correct

▲ Code Format

12.10. TRAILING WHITESPACE

Remove trailing whitespace at the end of each line of code. Omitting the closing PHP tag at the end of a file is preferred. There MUST be no trailing whitespace after statements or serial comma break or on blank lines.

✗ **Incorrect**: because there are two spaces after `$quotes_exist = false;`

```
9.  <?php
10.
11.         $quotes_exist = false;..
12.
13.         print_welcome_message();
14.
15.         // EOF
16.
```

Figure 86: Trailing Whitespace – Incorrect

✗ **Incorrect**: because there is a space after `,.`

```
12.         <?php
13.
14.         $my_movies = array(
15.         'Gone with the Wind',.
16.         'The Wizard of Oz',.
17.         'The Lord of the Rings',.
18.         'E.T. the Extra-Terrestrial'
19.         );
20.
21. // EOF
22.
```

Figure 87: Trailing Whitespace – Incorrect

✗ **Incorrect**: because the blank line below `$quotes_exist = false;` has two spaces.

```
9.  <?php
10.
11.         $quotes_exist = false;
12. ..
13.         print_welcome_message();
14.
15.         // EOF
16.
```

Figure 88: Trailing Whitespace – Incorrect

✓ **Correct**:

```
<?php

        $quotes_exist = false;

        print_welcome_message();

        // EOF

```

Figure 89: Trailing Whitespace – Correct

✓ **Correct**:

```
        <?php

        $my_movies = array(
                'Gone with the Wind',
                'The Wizard of Oz',
                'The Lord of the Rings',
                'E.T. the Extra-Terrestrial'
        );

// EOF

```

Figure 90: Trailing Whitespace – Correct

▲ Code Format

12.11. WHITESPACE INSENSITIVITY

Whitespace on the screen is usually not visible, including spaces, tabs, and end-of-line characters, such as carriage returns. This means many whitespace characters are the same as one whitespace character. A statement can essentially be broken up into several lines or a number of separate statements. Although whitespaces are generally ignored in PHP syntax, there are several places where you cannot have a space without affecting the sense (and the result). For instance, with a space in the middle of an operator, then it is no longer an operator. It is also true for numbers or in the middle of keywords, variables, constants, operators, and strings. Also, the opening parenthesis of the argument list or before the closing parenthesis. There MUST NOT be a space before each comma, and there MUST be one space after each comma.

In the following example spaces and tabs are used in a numeric operation, but in both cases, $four returns the same value.

Example:

```
7.  $four = 2 + 2; // single spaces
8.  $four <tab> = <tab2< tab> + <tab> 2 ; // spaces and tabs
9.  $four =
10. 2+
11. 2; // multiple lines
12.
```

Figure 91: Whitespace – Insensitivit – Example

The following two examples are the same:

Example: Advance whitespace insensitivity

```
17. <?php
18. function student_info
19.
20. $resource_name,
21. $role,
22. $company_name
23. )
24. {
25.                   echo 'The Name of resource is: $resource_name <br />';
26.     echo 'His role is: $role and his project is with: $company_name';
27. }
28. student_info(
29. 'Stephen Mitchell', 'Technical Writer', 'CSAB for DivTel'
30. )
31. // EOF
32.
```

Figure 92: Advance Whitespace – Insensitivity

Output:

```
13. The Name of the resource is: Stephen Mitchell
14. His role is: Technical Writer and his project is with: CSAB for DivTel
15.
```

Figure 93 Advance Whitespace – Insensitivity – Output

Example:

```
11. <?php
12.         // single space between $xyz, =, 11, +, 12
13.         $xyz = 11 + 12;
14.         echo $xyz .'<br />';
15.         // tabs and spaces
16.         $xyz =      11 12;
17.         echo $xyz;
18.
19.         // EOF
20.
```

Figure 94: Advance Whitespace – Insensitivity

Output:

```
16. 23
17.   23
```

Figure 95: Advance Whitespace – Output

12.12. SPACE USEAGE

ALWAYS put spaces after commas, and on both sides of logical, comparison, string and assignment operators.

Example:

```
18.       x === 23
19.       foo && bar
20.       ! foo
21.       array( 1, 2, 3 )
22.       $baz . '-5'
23.       $term .= 'X'
24.
```

Figure 96: Space Usage – Logical, Comparison, String, and Assignment Operators –Example

Ensure there are spaces between the opening and closing parentheses of if, elseif, foreach, for, and switch blocks.

Example:

```
25. foreach ( $foo as $bar ) { ...
26.
```

Figure 97: Space Usage – Spaces between the Opening and Closing Parentheses –Example

When defining a function, do it in the following manner:

Example:

```
27. function my_function( $param1, func_param( $param2 ) );
28.
29. Function my_other_function(); {  .
30.
```

Figure 98: Space – UsageDefining a Function –Example

When performing logical comparisons, do it in the following manner:

Example:

```
31. if ( ! $foo ) { ...
32.
```

Figure 99: Space Usage – Logical Comparisons –Example

Always use the short form of type cast: (int) rather than (integer) and (bool) instead of (boolean). For float casts use (float).:

Example:

```
33. foreach ( (array) $foo as $bar ) { ...
34.
35. $foo = (bool) $bar;
36.
```

Figure 100: Space Usage – Short Form of Type Cast –Example

When referring to array items, only include a space around the index if it is a variable.

Example:

```
37. $x = $foo['bar']; // correct
38. $x = $foo[ 'bar' ]; // incorrect
39.
40. $x = $foo[0]; // correct
41. $x = $foo[ 0 ]; // incorrect
42.
43. $x = $foo[ $bar ]; // correct
44. $x = $foo[$bar]; // incorrect
45.
```

Figure 101: Space Usage – Array Items –Example

When a case statement is used in a switch block, there must be no space before the colon.

Example:

```
46. switch ( $foo ) {
47.       case 'bar': // correct
48.       case 'ba' : // incorrect}
49.
```

Figure 102: Space Usage – Case Statement is Used in a Switch Block –Example

Similarly, there should be no space before the colon on return type declarations.

Example:

```
50. function sum( $a, $b ): float {
51.
52. return $a + $b;
53.
```

Figure 103: Space Usage – Return Type Declarations –Example

Unless otherwise specified, parentheses should have spaces inside of them.

Example:

```
54. if ( $foo && ( $bar || $baz ) ) { ...
55.
56. my_function( ( $x - 1 ) * 5, $y );
57.
```

Figure 104: Space Usage – Parentheses Should Have Spaces – Example

▲ Code Format

12.13. KEYWORDS

1. All PHP reserved keywords and types MUST be in lower case.
2. Any new types and keywords added to future PHP versions MUST be in lower case.
3. Short form of type keywords MUST be used, i.e., bool instead of boolean, int instead of integer, etc.

✗ **Incorrect**: `False`, `True`, and `NULL` aren't all lower-case.

```php
<?php

$is_true = FALSE;
$is_false = TRUE:
$movie_quote = NULL;

// EOF

```

Figure 105: Keywords – Incorrect

✓ **Correct**:

```php
<?php

$is_true = false;
$is_false = true:
$movie_quote = null;

// EOF

```

Figure 106: Keywords – Incorrect

▲ Code Format

12.14. STRINGS

In PHP, a string is a data type that allows you to store a series of characters. The PHP language only supports 256 characters as each letter is stored as a byte. A byte only has a max of 256 different values.

There are various ways that you can define a string within PHP. Each of these methods behaves differently and has its pros and cons.

These different methods of specifying a string are the following.

- Single Quotes (`' '`)
- Double Quotes (`" "`)
- Heredoc Syntax (`<<<IDENTIFIER`)
- Nowdoc Syntax (`<<<'IDENTIFIER'`)

The simplest way to specify a string is to enclose it in single quotes, `'the character'`.

✓ **Correct:**

```
1.  $suncom = 'A great project from a great team';
2.
```

Figure 107: String Names –Correct

If you want to add values from variables, concatenate strings. A space must be inserted before and after the dot for better readability.

✓ **Correct:**

```
1.  $message = 'The user,' . $name . ', signed up for the ' . $event . ' today!';
2.
```

Figure 108: String Names – Concatenatenation – Correct

Using the `dot` operator, you can break a string into multiple lines. You will have to indent each following line in order to mark it as part of the value assignment.

✓ **Correct:**

```
3.  $suncom = 'A great ' .
4.      'project from ' .
5.      'a great ' .
6.      'team';
7.
```

Figure 109: String Names – Value Assignment

▲ Code Format

12.15. VARIABLES

1. Each variable MUST be written in lower-case letters, and an underscore MUST separate every word.
2. A variable starts with the `$` sign, followed by the variable's name.
3. A variable name must start with a letter or the underscore character.
4. A variable name cannot start with a number.
5. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and `_`.
6. Variable names are case-sensitive: `$age` and `$AGE` are two different variables).

✗ **Incorrect**: since `$welcome_Message`, `$Welcome_Message`, and `$WELCOME_MESSAGE` are not all lower-case.

```
1. <?php
2.
3.  $welcome_Message = '';
4.  $Welcome_Message = '';
5.  $WELCOME_MESSAGE = '';
6.
7. // EOF
8.
```

Figure 110: Variables: – Incorrect

✗ **Incorrect**: because an underscore should separate `welcome` and `message`.

```
1.  <?php
2.
3.  $welcommessage = '';
4.
5.  // EOF
6.
```

Figure 111: Variables: – Incorrect

✓ **Correct**:

```php
<?php

  $welcome_message = '';

  // EOF

```

Figure 112: Variables: – Correct

See Also: Variable Initialization

See Also: External Variables

▲ Code Format

12.16. Global Variables

Usage of global variables should be kept to a minimum. Use Object-Oriented Programming instead.

Declare each global variable one variable per line, and MUST be indented after the first.

> **NOTE**:
>
> Usage of global variables should be kept to a minimum. Use OOP and factory patterns instead.

✗ **Incorrect**: `$app_config`, `$cache`, and `$db_connection` all appear on the same line.

```
1.  <?php
2.
3.  global $dms_config, $cache, $db_connection;
4.
5.  // EOF
6.
```

Figure 113: Global Variables – Incorrect

✗ **Incorrect**: since `$db_connection` and `$cache` are not indented once.

```
1.  <?php
2.
3.  global $dms_config,
4.  $cache,
5.  $db_connection;
6.
7.  // EOF
8.
```

Figure 114: Global Variables – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  global $dms_config,
4.  $cache,
5.  $db_connection;
6.
7.  // EOF
8.
```

Figure 115: Global Variables – Correct

See Also: Globals.

▲ Code Format

12.17. CONSTANTS

A constant is a name or an identifier for a simple value. A constant value cannot change during the execution of the script. By default, a constant is case-sensitive. By convention, constant identifiers MUST always be uppercase, in upper-case, and an underscore MUST separate each word.

> **NOTE**:
> Unlike variables, constants are automatically global across the entire script.

The visibility of a property, a method, or a constant can be defined by prefixing the declaration with the keywords `public`, `protected`, or `private`. Class members declared public can be accessed everywhere.

Visibility MUST be declared on all constants.

✕ **Incorrect**: since `Welcome_Message`, `Welcome_Message`, and `welcome_message` are not upper-case words.

```
1.  <?php
2.
3.  define('welcome_Message', '');
4.  define('Welcome_Message', '');
5.  define('welcome_message', '');
6.
7.  // EOF
8.
```

Figure 116: Constants – Incorrect

✕ **Incorrect**: The underscore SHOULD separate `WELCOME` from `MESSAGE`.

```
1.  <?php
2.
3.  define('WELCOMEMESSAGE', '');
4.
5.  // EOF
6.
```

Figure 117: Constants – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  define('WELCOME_MESSAGE', '');
4.
5.  // EOF
6.
```

Figure 118: Constants – Correct

See Also: Constant Names

▲ Code Format

12.18. STATEMENTS

PHP requires instructions to be terminated with a semicolon at the end of each statement. The closing tag of a block of PHP code automatically implies a semicolon; this is not necessary for terminating the last line of a PHP block.

Statements MUST be on a separate line and MUST end with a semicolon.

✗ **Incorrect**: because `print_welcome_message();` and `$quotes_exist = false;` are on the same line.

```
1.  <?php
2.
3.  $quotes_exist = false; print_welcome_message();
4.
5.  // EOF
6.
```

Figure 119: Statements – Incorrect

✗ **Incorrect**: because a semicolon is missing from `print_welcome_message()`.

```
1.  <div>
2.          <h1> <?= print_welcome_message() ?></h1>
3.  </div>
4.
```

Figure 120: Statements – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  $quotes_exist = false;
4.  print_welcome_message();
5.
6.  // EOF
7.
```

Figure 121: Statements – Correct

✓ **Correct**:

```
1. <div>
2.         <h1> <? = print_welcome_message(); ?></h1>
3. </div>
4.
```

Figure 122: Statements – Incorrect

▲ Code Format

12.19. OPERATORS

Make sure always to put spaces on both sides of the logical, comparison, string, assignment operators, and after the commas.

✗ **Incorrect**: because a space MUST surround the `=`, `+`, and `=` signs.

```php
<?php

$total = 3+14;
$string = 'Welcome, World! ';
$string.= 'Today is a great day!';

// EOF

```

Figure 123: Operators – Incorrect

✓ **Correct**:

```php
<?php

$total = 3 + 14;
$string = 'Hello, World! ';
$string = 'Today is a great day!';

// EOF

```

Figure 124: Operators – Correct

▲ Code Format

12.20. Unary Operators

Each unary operator MUST be attached to its corresponding variable or integer.

✗ **Incorrect**: because the `++` and `--` are separated by a space.

```
1.  <?php
2.
3.  $index ++;
4.  --$index;
5.
6.  // EOF
7.
```

Figure 125: Unary Operators – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  $index++;
8.  --$index;
4.
5.  // EOF
6.
```

Figure 126: Unary Operators – Correct

▲ Code Format

12.21. CONCATENATION PERIOD

There MUST be a space between the dot and the concatenated parts to improve readability.

Example:

```
1.  $string = 'Foo' . $bar;
2.  $string = $bar . 'foo';
3.  $string = bar() . 'foo';
4.  $string = 'foo' . 'bar';
5.
```

Figure 127: Concatenation Period – Example

When concatenating simple variables, you can use double quotes and add the variable inside; otherwise, use single quotes.

Example:

```
1.   $string = "Foo $bar";
2.
```

Figure 128: Concatenation Period – Simple Variables

When using the concatenating assignment operator `. =`, use a space on each side as with the assignment operator:

Example:

```
1.  $string  .  = 'Foo';
2.  $string  .  = $bar;
3.  $string  .  = baz();
4.
```

Figure 129 Concatenation Period– Assignment Operator – Example

✗ **Incorrect**: because a space is not present around the `. .`

```
1.  <?php
2.
3.  echo 'Hello, World! Today is '.$date.'!';
4.
5.  // EOF
6.
```

Figure 130: Concatenation Period – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  echo 'Hello, World! Today is ' . $date . '!';
4.
5.  // EOF
6.
```

Figure 131: Concatenation Period – Correct

▲ Code Format

12.22. Single Quotes

The single quote MUST be used. The use of double quotes is not recommended. As in the case of double quotes, `" "` double quote strings will display a host of escaped characters (including some regexes), and variables in the strings will be evaluated.

> **NOTE**:
> Single quotes inside of single quotes and double quotes inside of double quotes must be escaped

✗ **Incorrect**: since `"Hello, World!"` does not have single quotes.

```
1.  <?php
2.
3.  echo "Hello, World!";
4.
5.  // EOF
6.
```

Figure 132: Single Quotes – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  echo 'Hello, World!';
4.
5.  // EOF
6.
```

Figure 133: Single Quotes – Correct

▲ Code Format

12.23. Double Quotes

The issue of whether to use single quotes ' or double quotes " has been hotly debated. The single-quoted strings are not parsed, so whatever is put into the string will appear in the result. Strings enclosed in double quotes are parsed, and any PHP variables contained in the string will be evaluated.

The use of double quotes is not recommended.

1. If the string is enclosed in double-quotes ", PHP will interpret an assortment of escape sequences for special characters:
2. As in single-quoted strings, escaping any other character will also result in the backslash being printed.

✓ **Acceptable**: but variable `$date` is buried. Thus single quotes SHOULD be preferred.

```
1.  <?php
2.
3.  echo "Hello, World! Today is $date!";
4.
5.  // EOF
6.
```

Figure 134: Double Quotes – Acceptable

✓ **Acceptable**: when long pieces of text have apostrophes that would need to be escaped.

```
1.  <?php
2.
3.  echo "Hello, World! He\'s reading a book, and she\'s playing with kids. ";
4.
5.  // EOF
6.
```

Figure 135: Double Quotes – Acceptable

✓ **Preferred**:

```
1.  <?php
2.
3.  echo 'Hello, World! Today is ' . $date . '!';
4.
5.  echo 'Hello, World! He\'s reading a book, and she\'s playing with kids.';
6.
7.  // EOF
8.
```

Figure 136: Double Quotes – Preferred

▲ Code Format

12.24. BRACES

Braces are used around all statements, even single statements, when they are part of a control structure, such as an `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

In the style shown here, braces MUST be used for all blocks:

✓ **Correct**:

```
1.<?php
2.if ( condition )
3.{
4.    action1();
5.    action2();
6.}
7.elseif ( condition2 && condition3 )
8.{
9.    action3();
10.   action4();
11. }
12. else {
13.   defaultaction();
14. }
15. // EOF
16.
```

Figure 137: Brace Style – Correct

In order to reduce complexity, improve ease of testing, and increase readability, consider breaking up a long block into two or more shorter blocks, functions, or methods.

Always use braces, even if they aren't required.

✓ **Correct**:

```
1.<?php
2.if ( condition )
3.{
4.    action0();
5.}
6.if ( condition )
7.{
8.    action1();
9. }
1.elseif ( condition2 )
10.  {
11.     action2a();
12.     action2b();
13.                  }
2.Foreach ( $items as $item )
3.{
14. process_item( $item );
15. }
16. // EOF
17.
```

Figure 138: Brace Style – Correct

The requirement for braces simply means that single-statement inline control structures are not permitted. In templates containing PHP code embedded within HTML, you are free to use alternative syntax for control structures (i.e. if/endif, while/endwhile):

✓ **Correct**:

```
1.  <?php if ( have_posts() ) : ?>
2.  <div class = 'hfeed'>
3.  <?php while ( have_posts() ) : the_post(); ?>
4.          <article id = 'post-<?php the_ID() ?>' class ='<?php post_class() ?>'>
5.          <!-- ... -->
6.          </article>
7.  <?php endwhile; ?>
8.  </div>
9.  <?php endif; ?>
10. ?>
```

Figure 139: Brace Style – Correct

▲ Code Format

12.25. FORMATTING SQL STATEMENTS

When formatting SQL statements, you may break them into several lines and indent if it is sufficiently complex to warrant it. Most statements work well as one line, though. Always capitalize the SQL parts of the statement like UPDATE or WHERE.

Functions that update the database should expect their parameters to lack SQL slash escaping when passed. Escaping should be done as close to the time of the query as possible, preferably by using `$dmsdb->prepare()`.

`$dmsdb ->prepare()` is a method that handles escaping, quoting and int-casting for SQL queries. It uses a subset of the `sprintf()` style of formatting.

Example:

```
1.  $var = "'dangerous'"; // raw data that may or may not need to be escaped
2.  $id = some_foo_number(); // data we expect to be an integer, but we're not certain
3.
4.  $wpdb->query( $dmsdb->prepare( 'UPDATE $wpdb->posts SET post_title = %s WHERE ID = %d',
    $var, $id ) );
5.
```

Figure 140; Formatting SQL Statements – Example

`%s` is used for string placeholders, and `%d` is used for integer placeholders. Note that they are not 'quoted'. `$dmsdb->prepare()` will take care of escaping and quoting for us. The benefit of this is that it is easy to see at a glance whether something has been escaped or not because it happens right when the query happens.

Example:

```
1.  $var = "'dangerous'"; // raw data that may or may not need to be escaped
2.  $id = some_foo_number(); // data we expect to be an integer, but we're not certain
3.
4.  $wpdb->query( $wpdb->prepare( 'UPDATE $wpdb->posts SET post_title = %s WHERE ID
    = %d', $var, $id ) );
5.
```

Figure 141: Formatting SQL Statements – Example

String placeholders are represented by %s and integer placeholders by %d. Note that they are not 'quoted'! $dmsdb->prepare() will take care of escaping and quoting for us. In this way, we are not required to remember to use esc_sql(), and it is also easy to see at a glance whether something has been escaped or not since it happens during the query.

12.25.1. Database Queries

Avoid directly touching the database. If a defined function can provide you with the data you need, use it. It is always better to use a function that has been defined to get the data you need. It can be much faster to use database abstractions (functions rather than queries) when results are cached in memory, which helps keep your code forward-compatible.

▲ Formatting SQL statements

12.26. Clever Code

In general, readability is more important than cleverness or brevity.

Example:

```
1.  isset( $var ) || $var = some_function();
```

Figure 142: Code Format – Clever Code –Example

The above line is clever, but it takes a while to grasp if you are unfamiliar with it. So, just write it as follows:

Example:

```
1.  if ( ! isset( $var ) ) {
2.  $var = some_function();
3.  }
4.
```

Figure 143: Code Format – Clever Code –Example

Unless absolutely necessary, loose comparisons should not be used, as their behaviour can be misleading.

Incorrect:

```
1.  if ( 0 == strpos( 'MyFlorida', 'foo' ) ) {
2.  echo __( 'Yay MyFlorida!' );
3.  }
4.
```

Figure 144: Code Format – Clever Code – Incorrect

Correct:

```
1.  if ( 0
2.  strpos( 'MyFlorida', 'foo' ) ) {
3.  echo __( 'Yay MyFlorida!' );
4.  }
5.
```

Figure 145: Code Format – Clever Code –Correct

Assignments must not be placed in conditionals.

Incorrect:

```
1.  if ( $data = $wpdb->get_var( '...' ) ) {
2.  // Use $data
3.  }
4.
```

Figure 146: Clever Code – Incorrect

✓ **Correct:**

```
1.   $data = $wpdb->get_var( '...' );
2.   if ( $data ) {
3.   // Use $data
4.   }
5.
```

Figure 147: Clever Code – Correct

In a switch statement, it's okay to have multiple empty cases fall through to a common block. If a case contains a block, then it falls through to the next block; however, this must be explicitly commented.

➕ **Example:**

```
1.  switch ( $foo ) {
2.      case 'bar'              // Correct, an empty case can fall through without
3.                              comment.
4.      case 'baz':
5.          echo $foo           // Incorrect, a case with a block must break, return
                                        or have a comment.
6.      case 'cat':
7.          echo 'mouse';
8.          break               // Correct, a case with a break does not require a
                                        comment.
9.  case 'dog':
10.         echo 'horse';
11.         // no break         // Correct, a case can have a
    comment                                             to explicitly indent the fall
    through.
12.         echo 'bird';
13.         break;
14. }
15.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

13. FUNCTIONS

There should be no spaces between the function name and the opening parenthesis, nor between `this` and the function's first parameter; each parameter should be separated by a comma (if present), and there should be no space between the last parameter and the closing parenthesis. The `equal` sign should be preceded by a space followed by exactly one space. Tab alignment over multiple lines is permitted.

Correctly using functions confers:

- Better code organization – PHP functions allow us to group blocks of related code that perform a specific task together
- Reusability – once defined, a function can be called by a number of scripts in PHP files
- Easy maintenance- updates to the system only need to be made in one place

This section describes the functions' names, calls, arguments, and declarations.

1. Function Name must start with a letter or underscore — not a number. (Starting function names with an underscore are typically reserved for super global. Separate words with an underscore, and all letters MUST be in lower case.
   - e.g., `function welcome_message() {`

   See Also: Function Names

2. Function Prefix Begin with the verb.
   - e.g., `get_`, `add_`, `update_`, `delete_`, `convert_`, etc.

   See Also: Function Prefix

3. Function Call A space MUST NOT appear between the function name and the open parenthesis.
   - e.g., `func();`
   - Calling functions must always start with the keyword function
   - PHP code must be contained within curly brackets `{ }`

4. Function Definitions

Function definitions start on a new line, and the opening and closing braces are also placed on new lines. An empty line should precede lines specifying the return value.

Function definitions must include a documentation comment in accordance with the Commenting section of this document.

- Short description (required, followed by a blank line)
- Long description (optional, followed by a blank line)
- @param (required if there are method or function arguments, a blank line follows the last @param tag)
- @return (required, followed by a blank line)
- All other tags in alphabetical order, however @since is always required

Example:

```
1.  <span class = 'comment'>/**
2. * A utility class.
3. *
4. * @package     Zend Framework
5. * @subpackage  XBase
6. *
```

```
7. * @param       string  $path  The library path in dot notation.
8. *
9. * @return      void
10.  *
11.  * @since       1.6
12.  */</span>
13. <span class = 'storage function'>function</span> <span class ='entity name
    function'>jimport</span>(<span class = 'variable dollar-sign'>$</span><span class =
    'variable'>path</span>)
14. {
15.     <span class = 'comment'>// Body of method.</span>
16. }
17.
```

Figure 172: Function Definition – Example

If a function definition goes over multiple lines, all lines must be indented with one tab, and the closing brace must go on the same line as the last parameter.

Example:

```
1.function fooBar($param1, $param2,
2.    $param3, $param4)
3.{
4.    // Body of method.
5.}
6.
```

Figure 173: Function Definition – Example

```
16. ▲       switch ( $foo ) {
17.                 case 'bar'                  // Correct, an empty case can fall
    through without
18.                                         comment.
19.                 case 'baz':
20.                         echo $foo // Incorrect, a case with a block must break,
    return                                              or have a comment.
21.                 case 'cat':
22.                         echo 'mouse';
23.                         break     // Correct, a case with a break does not require a
                                      comment.
24.     case 'dog':
25.         echo 'horse';
26.         // no break         // Correct, a case can have a
    comment                                             to explicitly indent the fall
    through.
27.         echo 'bird';
28.         break;
29. }
30.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

5. Function Arguments
   - A space MUST NOT appear before a comma
   - A space MUST follow the comma
   - Line breaks MAY be used for long arguments
   - Each argument MUST be on its own line
   - Each argument MUST be indented once
   - Order MUST be from required to optional first
   - Order MUST be from high to low importance second
   - Defaults MUST be descriptive
   - Type hinting MUST be used
   - e.g., `func($arg1, $arg2 = 'asc', $arg3 = 100);`

6. Function Declaration Documentation MUST be provided using and should include.
    - A brief description
    - If needed, provide a long description
    - `@access`: Assuming public, private, or protected
    - `@author`: Author name
    - `@global`: if applicable, global variables function uses
    - `@param`: Parameters with a data type, variable name, and description
    - `@return`: Return data type, if applicable
7. Function Return
    - MUST occur as early as possible
    - MUST be previously initialized at the top
    - Must be preceded by a blank line, except for inside a control statement
        - i.e., `if (!$expr) { return false; }`

▲

13.1. FUNCTION NAME

The same rules apply to function names as to other labels in PHP. The name of a valid function should start with a letter or underscore and then may include any number of letters, numbers, or underscores.

When naming functions, variables, or classes, keep these considerations in mind:

1. Choose a word with meaning (provide some context).
2. Avoid generic names (like `tmp`).
3. Attach additional information to a name (use suffix or prefix).
4. Don't make your names too long or too short.
5. Use consistent formatting.

All function names MUST be in lower-case, with underscores separating words.

✕ **Incorrect**: because the function names are not all lower-case.

```php
<?php

get_welcome_message();
Get_welcome_message();
GET_WELCOME_MESSAGE();
// EOF

```

Figure 149: Function Names – Incorrect

✕ **Incorrect**: since `get`, `welcome`, and `message` does not have an underscore between them.

```php
<?php

getwelcomemessage();

// EOF

```

Figure 150: Function Names – Incorrect

See Also: Function Names

✓ **Correct**:

```
1.  <?php
2.
3.  get_welcome_message();
4.
5.  // EOF
6.
```

Figure 151: Function Names – Correct

```
31. ▲       switch ( $foo ) {
32.                     case 'bar'                    // Correct, an empty case can fall
    through without
33.                                         comment.
34.                     case 'baz':
35.                                 echo $foo // Incorrect, a case with a block must break,
    return                                                  or have a comment.
36.                     case 'cat':
37.                                 echo 'mouse';
38.                                 break     // Correct, a case with a break does not require a
                                              comment.
39.     case 'dog':
40.         echo 'horse';
41.         // no break         // Correct, a case can have a
    comment                                                 to explicitly indent the fall
    through.
42.         echo 'bird';
43.         break;
44. }
45.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.2. FUNCTION PREFIX

The function prefix MUST begin with a verb.

Naming functions with prefix verb.

A/HC/LC Pattern

When naming functions, there is a useful pattern to follow:

prefix? + action (A) + high context (HC) + low context? (LC).

The following table illustrates how this pattern can be applied.

Name	Prefix	Action (A)	High context (HC)	Low context (LC)
getUser		get	User	
getUserMessages		get	User	Messages
handleClickOutside		handle	Click	Outside
shouldDisplayMessage	should	Display	Message	

*Table 5: Function Prefix – **A/HC/LC** Pattern*

NOTE:

Variables take on different meanings based on their context. `ShouldUpdateComponent`, for example, indicates that you are about to update a component, whereas `shouldUpdateComponent` indicates that the component will update on its own, and you are only controlling when it should be updated. To put it another way, high context emphasizes the significance of a function or variable.

Actions:

This is the verb part of your function name. It provides an explanation of what the function does.

get:

Immediately accesses data (i.e., shorthand getter of internal data).

Example:

```
1.  function getFruitCount() {
2.    return .this.fruits.length
3.  }
4.
```

Figure 152: Function Prefix – get –Example

See Also: compose:

set:

The value `A` is set to a value `B` in a declarative manner.

Example:

```
1.  let fruits : 0
2.
3.  Function setFruits(nextFruits) {
4.      fruits = nextFruits
5.        }
6.
7.        setFruits(5)
8.        console..log(fruits) // 5
9.
```

Figure 153: Function Prefix – set –Example

reset:

Resets a variable to its initial value or state.

Example:

```
1.        const initialFruits = 5
2.        let fruits = initialFruits
3.        setFruits(10)
4.        console..log(fruits) // 10
5.
6.        function resetFruits() {
7.     fruits = initialFruits
8.        }
9.        resetFruits()
10.   console.log(fruits) // 5
11.
```

Figure 154: Function Prefix – reset –Example

fetch:

Request for data, which will take some indeterminate amount of time (i.e., async request).

Example:

```
1.        function fetchPosts(postCount) {
2.    .return fetch('https://api.dev/posts', {...})
3.        }
4.
```

Figure 155: Function Prefix – fetch –Example

remove:

Removing something from somewhere.

As an example, if you have a set of selected filters on a search page, removing a particular filter from the collection is `removeFilter`, not `deleteFilter` (and you would naturally say this in English as well):

Example:

```
1.        function removeFilter(filterName, filters)
2.  {
3.    .return filters.filter((name) => name !== filterName)
4.        }
5.
6.        const selectedFilters = ['price', 'availability', 'size']
7.        removeFilter('price', selectedFilters)
8.
```

Figure 156: Function Prefix – remove –Example

See Also: delete:

delete:

Erases something completely from existence.

Example:

```
1.        function deletePost(id)
2.  {
3.    return database.find({ id }).delete()
4.        }
5.
```

Figure 157: Function Prefix – delete –Example

See Also: remove:

compose:

From existing data, creates a new one. Usually applicable with strings, objects, and functions.

Example:

```
1.        function composePageUrl(pageName, pageId) {
2.    .return (pageName.vLowerCase() + '-' + pageId)
3.        }
4.
```

Figure 158: Function Prefix – compose –Example

See Also: Naming functions with prefix verb.

A/HC/LC Pattern

When naming functions, there is a useful pattern to follow:

prefix? + action (A) + high context (HC) + low context? (LC).

The following table illustrates how this pattern can be applied.

Name	Prefix	Action (A)	High context (HC)	Low context (LC)
getUser		get	User	
getUserMessages		get	User	Messages
handleClickOutside		handle	Click	Outside
shouldDisplayMessage	should	Display	Message	

*Table 5: Function Prefix – **A/HC/LC** Pattern*

NOTE:

Variables take on different meanings based on their context. `ShouldUpdateComponent`, for example, indicates that you are about to update a component, whereas `shouldUpdateComponent` indicates that the component will update on its own, and you are only controlling when it should be updated. To put it another way, high context emphasizes the significance of a function or variable.

Actions:

This is the verb part of your function name. It provides an explanation of what the function does.

get:

handle:

Handles an action. Most commonly used as a name for a callback method.

Example:

```
1.        function handleLinkClick() {
2.    console..log('Clicked a link!')
3.        }
4.  link.addEventListener('click', handleLinkClick)
5.
```

Figure 159: Function Prefix – handle –Example

Context:

Functions are often actions performed on something. It is important to describe its operable domain, or at least an expected data type.

Example:

```
1.  /* A pure function operating with primitives */
2.  function filter(list, predicate) {
3.    return list.filter(predicate)
4.  }
5.  /* Function operating exactly on posts */
6.  function getRecentPosts(posts) {
7.    return filter(posts, (post) => post.date === Date.now())
8.  }
9.
```

Figure 160: Function Prefix – handle –Example

Prefixes:

The prefix enhances the meaning of a variable. It is seldom used in function names.

is:

Provides some information about the current state or context (typically boolean).

Example:

```
1.  const color = 'blue'
2.  const isBlue = color === 'blue' // characteristic
3.  const isPresent = true // state
4.
5.  if (isBlue && isPresent) {
6.    console.log('Blue is present!')
7.        }
8.
```

Figure 161: Function Prefix – is –Example

has:

Describes whether the current context contains a particular value or state (usually a boolean).

Example:

```
1.  /* Bad */
2.  const isProductsExist = productsCount > 0
3.  const areProductsPresent = productsCount > 0
4.
5.  /* Good */
6.  const hasProducts = productsCount > 0
7.
```

Figure 162: Function Prefix – has –Example

should:

Reflects a positive conditional statement (usually boolean) along with a certain action.

Example:

```
function shouldUpdateUrl(url, expectedUrl) {
  return url !== expectedUrl
}

```

Figure 163: Function Prefix – should –Example

min/max:

Specifies a minimum or maximum value. Used to describe boundaries or limits.

Example:

```
/**
* Renders a random amount of posts within
* the given min/max boundaries.
*/
function renderPosts(posts, minPosts, maxPosts) {
  return posts.slice(0, randomBetween(minPosts, maxPosts))
      }

```

Figure 164: Function Prefix – min/max –Example

prev/next:

Describes the previous or next state of a variable in the current context. It is used to describe state transitions.

Example:

```
function fetchPosts() {
  const prevPosts = .this.state.posts
  const fetchedPosts = fetch('...')
  const nextPosts = concat(prevPosts, fetchedPosts)
  this.setState({ posts: nextPosts })
}

```

Figure 165: Function Prefix – prev/next –Example

See Also: Function Names

```
▲       switch ( $foo ) {
                case 'bar'                    // Correct, an empty case can fall through without
                                          comment.
                case 'baz':
                          echo $foo // Incorrect, a case with a block must break, return
                                                    or have a comment.
                case 'cat':
```

```
52.                             echo 'mouse';
53.                             break     // Correct, a case with a break does not require a
                                          comment.
54.     case 'dog':
55.         echo 'horse';
56.         // no break         // Correct, a case can have a
    comment                                                 to explicitly indent the fall
    through.
57.         echo 'bird';
58.         break;
59. }
60.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

Singular and Plurals:

Like a prefix, variable names can be made singular or plural depending on whether they hold a single value or multiple values.

Example:

```
1.  /* Bad */
2.
3.  const friends = 'Rhonda'
4.  const friend = ['Nelson', 'Sean', 'Todd', 'Raghib.']
5.  /* Good */
6.  const friend  = ' Rhonda '
7.  const friends = [' Nelson ', ' Sean ', ' Todd ', 'Raghib.']
8.
```

Figure 166: Function Prefix – Singular and Plurals –Example

Incorrect: because no verb is prefixed to functions.

```
1.  <?php
2.
3.  active_users();
4.  network_location($location1, $location2);
5.  widget_form($id);
6.
7.  // EOF
8.
```

Figure 167: Function Prefix – Incorrect

Correct:

```
1.  <?php
2.
3.  get_active_users();
4.  move_network_location($location1, $location2);
5.  delete_widget_form($id);
6.
7.  // EOF
8.
```

Figure 168: Function Prefix – Correct

```
61. ▲       switch ( $foo ) {
62.                 case 'bar'                  // Correct, an empty case can fall
    through without
63.                                     comment.
64.                 case 'baz':
65.                         echo $foo // Incorrect, a case with a block must break,
    return                                              or have a comment.
66.                 case 'cat':
67.                         echo 'mouse';
68.                         break     // Correct, a case with a break does not require a
                                      comment.
69.     case 'dog':
70.         echo 'horse';
71.         // no break        // Correct, a case can have a
    comment                                             to explicitly indent the fall
    through.
```

```
72.         echo 'bird';
73.         break;
74. }
75.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.3. FUNCTION CALL

There should be no spaces between the function name and the opening parenthesis when calling functions. Likewise, there should be no space between this and the first parameter. There should be a space after the comma between each parameter (if they are present), and there should be no space between the last parameter and the closing parenthesis. There should be exactly one space before and one space after with the equals sign. Tab alignment over multiple lines is allowed.

+ Example:

```
1.  $var = foo($bar, $baz, $quux);
```

Figure 169: Function Call – Example

✗ **Incorrect**: because a space appears between `get_welcome_message` and `( )`.

```
1.  <?php
2.
3.  print_welcome_message ();
4.
5.  // EOF
6.
```

Figure 170: Function Call – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  print_welcome_message();
4.
5.  // EOF
6.
```

Figure 171: Function Call – Correct

```
76. ▲       switch ( $foo ) {
77.                     case 'bar'                      // Correct, an empty case can fall
    through without
78.                                         comment.
79.                     case 'baz':
80.                             echo $foo // Incorrect, a case with a block must break,
    return                                                  or have a comment.
81.                     case 'cat':
82.                             echo 'mouse';
83.                             break     // Correct, a case with a break does not require a
                                          comment.
84.     case 'dog':
85.         echo 'horse';
86.         // no break         // Correct, a case can have a
    comment                                                 to explicitly indent the fall
    through.
87.         echo 'bird';
88.         break;
```

```
89. }
90.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.4. FUNCTION DEFINITIONS

Function definitions start on a new line, and the opening and closing braces are also placed on new lines. An empty line should precede lines specifying the return value.

Function definitions must include a documentation comment in accordance with the Commenting section of this document.

- Short description (required, followed by a blank line)
- Long description (optional, followed by a blank line)
- @param (required if there are method or function arguments, a blank line follows the last @param tag)
- @return (required, followed by a blank line)
- All other tags in alphabetical order, however @since is always required

Example:

```
18.         <span class = 'comment'>/**
19.  * A utility class.
20.  *
21.  * @package     Zend Framework
22.  * @subpackage  XBase
23.  *
24.  * @param       string  $path  The library path in dot notation.
25.  *
26.  * @return      void
27.  *
28.  * @since       1.6
29.  */</span>
30. <span class = 'storage function'>function</span> <span class ='entity name
    function'>jimport</span>(<span class = 'variable dollar-sign'>$</span><span class =
    'variable'>path</span>)
31. {
32.     <span class = 'comment'>// Body of method.</span>
33. }
34.
```

Figure 172: Function Definition – Example

If a function definition goes over multiple lines, all lines must be indented with one tab, and the closing brace must go on the same line as the last parameter.

Example:

```
function fooBar($param1, $param2,
    $param3, $param4)
{
    // Body of method.
}
```

Figure 173: Function Definition – Example

```
    switch ( $foo ) {
                case 'bar'                  // Correct, an empty case can fall
    through without
                                        comment.
                case 'baz':
                        echo $foo // Incorrect, a case with a block must break,
    return                                          or have a comment.
                case 'cat':
                        echo 'mouse';
                        break     // Correct, a case with a break does not require a
                                  comment.
    case 'dog':
                        echo 'horse';
                // no break         // Correct, a case can have a
    comment                                         to explicitly indent the fall
    through.
                        echo 'bird';
                        break;
        }
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.5. FUNCTION ARGUMENTS

Function arguments are the values passed into a function. An argument is a variable inside the function.

- Arguments are defined inside the parentheses, which are there immediately after the function name
- A function can have any number of arguments; separate the arguments with commas
- An argument name should obey the same rules as a variable since arguments are variables
- A function MUST NOT have a space before the comma
- A function MUST have a space after the comma
- A function MAY use line breaks for long arguments
- A function MUST then place each argument on its own line
- A function MUST then indent each argument once
- A function MUST be ordered from required to optional first
- A function MUST be ordered from high to low importance second
- A function MUST use descriptive defaults
- A function MUST use type hinting

✗ **Incorrect**: because the space before , is not allowed.

```php
<?php

my_function($arg1 , $arg2 , $arg3);

// EOF

```

Figure 174: Function Arguments – Incorrect

✗ **Incorrect**: because there SHOULD be no space after ,.

```php
<?php

my_function($arg1,$arg2,$arg3);

// EOF

```

Figure 175: Function Arguments – Incorrect

✕ **Incorrect:** because `$arg1_with_a_long_name` SHOULD be on its own line.

```
1.  <?php
2.
3.  my_other_function($arg1_with_a_really_long_name,
4.  $arg2_also_has_a_long_name,
5.  $arg3
6.  );
7.
8.  // EOF
9.
```

Figure 176: Function Arguments – Incorrect

✕ **Incorrect**: since the arguments have not been indented once.

```
1.  <?php
2.
3.  my_other_function(
4.  $arg1_with_a_really_long_name,
5.  $arg2_also_has_a_long_name,
6.  $arg3
7.  );
8.
9.  // EOF
10.
```

Figure 177: Function Arguments – Incorrect

✕ **Incorrect**: because `$type`, `$order`, and `$limit` are not in order of required to optional as they SHOULD be.

```
1.  <?php
2.
3.  function get_objects($limit, $order, $type) {
1.          // ...
4.  }
5.
6.  // EOF
7.
```

Figure 178: Function Arguments – Incorrect

✕ **Incorrect**: because `$limit`, `$order`, and `$type` are not ordered by importance.

```
1.  <?php
2.
3.  function get_objects($limit, $order, $type) {
2.          // ...
4.  }
5.
6.  // EOF
7.
```

Figure 179: Function Arguments – Incorrect

✗ **Incorrect**: since `true` is not a descriptive default for `$order`.

```php
<?php

function get_objects($type, $order = true, $limit = 100) {
        // ...
}

// EOF

```

Figure 180: Function Arguments – Incorrect

✗ **Incorrect**: because `$users` and `$office` do not have their data types.

```php
<?php

function add_users_to_office($users, $office) {
        // ...
}

// EOF

```

Figure 181: Function Arguments – Incorrect

✓ **Correct**:

```php
<?php

my_function($arg1, $arg2, $arg3);

my_other_function(
        $arg1_with_a_really_long_name,
        $arg2_also_has_a_long_name,
        $arg3
);

function get_objects($type, $order = 'asc', $limit = 100) {
    // ...
}
function add_users_to_office(array $users, Office $office) {
        // ...
}
// EOF

```

Figure 182: Function Arguments – Correct

```php
        switch ( $foo ) {
                case 'bar'                      // Correct, an empty case can fall through without
                                                comment.
                case 'baz':
                        echo $foo // Incorrect, a case with a block must break, return
                                                or have a comment.
                case 'cat':
                        echo 'mouse';
                        break     // Correct, a case with a break does not require a
                                                comment.
```

```
114.                 case 'dog':
115.                         echo 'horse';
116.                 // no break         // Correct, a case can have a
    comment                                                 to explicitly indent the fall
    through.
117.                         echo 'bird';
118.                         break;
119.         }
120.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.6. FUNCTION DECLARATION

Function declaration MUST be documented and SHOULD include:

- Short description
- Optional long description, if needed
  - `@access`: private or protected (assumed public)
  - `@author`: Author name
  - `@global`: Global variables function uses, if applicable
  - `@param`: Parameters with a data type, variable name, and description
  - `@return`: Return data type, if applicable

```
121.        ▲       switch ( $foo ) {
122.                    case 'bar'                  // Correct, an empty case can
    fall through without
123.                                                comment.
124.                    case 'baz':
125.                        echo $foo // Incorrect, a case with a block must
    break, return                                   or have a comment.
126.                    case 'cat':
127.                        echo 'mouse';
128.                        break     // Correct, a case with a break does not
    require a                                       comment.
129.                case 'dog':
130.                    echo 'horse';
131.                // no break         // Correct, a case can have a
    comment                                         to explicitly indent the fall
    through.
132.                    echo 'bird';
133.                    break;
134.        }
135.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

✗ **Incorrect**: because there is no documentation for `my_function`.

```
1.  <?php
2.
3. function my_function($id, $type, $width, $height) {
4.  // ...
5.  return $Photo;
6.  }
7.
8.  // EOF
9.
```

Figure 183: Function Declaration – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  /**
4.  * Get a photo from the user
5.  *
6.  * Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut id volutpat
7.  * orci. Etiam pharetra eget turpis non ultrices. Pellentesque vitae risus
8.  * sagittis, vehicula massa eget, tincidunt ligula.
9.  *
10.  * @access private
11.  * @author Firstname Lastname
12.  * @global object $post
13.  * @param int $id Author ID
14.  * @param string $type Type of photo
15.  * @param int $width Photo width in px
16.  * @param int $height Photo height in px
17.  * @return object Photo
18.  */
19. function my_function($id, $type, $width, $height) {
20.                     // ...
21.                             return $Photo;
22. }
23.
24. // EOF
25.
```

Figure 184: Function Declaration – Correct

```
136.        ▲           switch ( $foo ) {
137.                            case 'bar'                  // Correct, an empty case can
    fall through without
138.                                                        comment.
139.                            case 'baz':
140.                                    echo $foo // Incorrect, a case with a block must
    break, return                                           or have a comment.
141.                            case 'cat':
142.                                    echo 'mouse';
143.                                    break     // Correct, a case with a break does not
    require a                                               comment.
144.                    case 'dog':
145.                                    echo 'horse';
146.                    // no break             // Correct, a case can have a
    comment                                                 to explicitly indent the fall
    through.
147.                            echo 'bird';
148.                            break;
```

```
149.        }
150.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.7. FUNCTION RETURN

Function return:

- MUST occur as early as possible
- MUST be initialized prior at top
- MUST be preceded by a blank line, except inside the control statement

✗ **Incorrect**: since `get_object` does not return as soon as possible.

```
1.  <?php
2.
3.  function get_object() {
4.  $var = false;
5.  if($expr1) {
6.          // ...
7.  `       if($expr2) {
8.                  // ...
9.          }
10.         }
11.
12.         return $var;
13. }
14.
15. // EOF
16.
```

Figure 185: Function Return – Incorrect

✗ **Incorrect**: since `$movies` is not initialized at the top.

```
1.  <?php
2.
3.  function get_movies() {
4.          // ...
5.
6.          return $movies;
7.}
8.
9.  // EOF
10.
```

Figure 186: Function Return – Incorrect

✗ **Incorrect**: because there is no blank line before `return $movies`.

```php
<?php

function get_movies() {
        $movies = array();
// ...
        return $movies;
}

// EOF

```

Figure 187: Function Return – Incorrect

✓ **Correct**:

```php
<?php

function get_object() {
    $var = false;
    if (!$expr1) {
        return $var;
}
    if (!$expr2) {
        return $var;
    }
    // ...

    return $var;
}

// EOF

```

Figure 188: Function Return – Correct

✓ **Correct**:

```php
<?php

function get_movies() {
    $movies = array();
    // ...

    return $movies;
}

// EOF

```

Figure 189: Function Return – Correct

```
        ▲       switch ( $foo ) {
                    case 'bar'                  // Correct, an empty case can
    fall through without
                                                comment.
                    case 'baz':
                        echo $foo // Incorrect, a case with a block must
    break, return                                   or have a comment.
                    case 'cat':
                        echo 'mouse';
```

```
158.                                    break     // Correct, a case with a break does not
    require a                                                 comment.
159.                case 'dog':
160.                          echo 'horse';
161.                // no break         // Correct, a case can have a
    comment                                                   to explicitly indent the fall
    through.
162.                          echo 'bird';
163.                          break;
164.        }
165.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

13.8. Type Hinting

Function and method parameters should specify their data types using type hinting whenever possible. Because type hinting cannot be used with scalar types such as int or string, this only occurs when arguments are arrays or objects.

Example:

```
1.  class MyClass {
2.      /**
3.      * First parameter must be an object implementing the OtherClassInterface.
4.      */
5.      public function test(OtherClassInterface $otherClass)
6.      {
7.        echo $otherClass->var;
8.      }
9.
10.
11.     /**
12.     * First parameter must be an array.
13.     */
14.     public function test_array(array $input_array) {
15.       print_r($input_array);
16. }
```

Figure 190: Type Hinting – Function and Method Parameters

It is recommended to specify the particular interface the class implements rather than the name of the class when using type hinting on classes. Other classes that implement the same interface can also be passed in, allowing easier swappability.

```
166.        ▲       switch ( $foo ) {
167.                    case 'bar'                  // Correct, an empty case can
    fall through without
168.                                                comment.
169.                    case 'baz':
170.                        echo $foo // Incorrect, a case with a block must
    break, return                                   or have a comment.
171.                    case 'cat':
172.                        echo 'mouse';
173.                        break     // Correct, a case with a break does not
    require a                                       comment.
174.                case 'dog':
175.                    echo 'horse';
176.                // no break         // Correct, a case can have a
    comment                                         to explicitly indent the fall
    through.
177.                    echo 'bird';
178.                    break;
179.        }
180.
```

Figure 148: Clever Code –Switch Statement –Example

▲ Code Format

Functions

14. Control Structures

This section describes how control structures are laid out and how to use them.

In order to discern control statements from function calls, there should be one space between the control keyword and the opening parenthesis so that control statements can be easily distinguished.

In general, control structures should follow the following rules:

- **Keyword**: There MUST be one space after the control structure keyword
- **Opening parenthesis**: There MUST NOT be a space after the opening parenthesis
- **Closing parenthesis**: There MUST NOT be a space before the closing parenthesis
- **Opening brace**: There MUST be one space between the closing parenthesis and the opening brace
- **Structure body**: The structure body MUST be indented once
- **Closing brace**: The closing brace MUST be on the following line after the body
- **Nesting**: MUST NOT exceed three levels

Always use curly braces, even if they are technically optional. They increase readability and reduce the likelihood of logic errors when new lines are added.

Structures MUST be enclosed by braces.

Some control structures have additional requirements in addition to the rules above:

1. If, Elseif, Else
   - `elseif` MUST be used instead of `else if`
   - elseif and else MUST be between `}` and `{` on one line
2. Switch, Case
   - Case statement MUST be indented once
     - i.e., ⇥ `case 1:`
   - Case body MUST be indented twice
     - i.e., ⇥ ⇥ `func();`

- Break keyword MUST be indented twice
  - i.e., ⇥ ⇥ `break;`
- Case logic MUST be separated by one blank line
  - i.e., `case 1: .. break; ↵ ↵ case 2: .. break;`

3. While, Do While

```php
<?php

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

// EOF

```

Figure 202: Control Structures – do while–Example

▲ Control Structures

4. For, Foreach
5. Try, Catch
    - catch MUST be between } and { on one line

▲

14.1. IF, ELSEIF, ELSE

1. The keyword `elseif` SHOULD be used instead of `else if` so that all control keywords look like single words.
2. The else if and else if conditions MUST be on a single line between `}` and `{`.

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.

+ Example:

```php
<?php

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

// EOF
```

Figure 191:Control Structures – if, elseif, else – Example

✕ **Incorrect**: because Instead of `elseif`, `else if` was used.

```php
<?php

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

// EOF

```

Figure 192: Control Structures – if, elseif, else – Incorrect

✗ **Incorrect**: because there is no `elseif` and `else` between `}` and `{` on a single line.

```
1.  <?php
2.
3.  if ($expr1) {
4.          // if body
5.  }
6.  elseif ($expr2) {
7.          // elseif body
8.  }
9.  else {
10.         // else body
11. }
12.
13. // EOF
14.
```

Figure 193: Control Structures – if, elseif, else – Incorrect

✗ **Incorrect**: since curly braces do not surround the structure body.

```
1.  <?php
2.
3.  $result1 = if ($expr1) ? true : false;
4.
5.  if($expr2)
6.          $result2 = true;
7.
8.  // EOF
9.
```

Figure 194: Control Structures – if, elseif, else – Incorrect

✓ **Correct**:

```php
<?php

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

if ($expr1) {
        $result1 = true;
} else {
        $result1 = false;
}

if ($expr2) {
        $result2 = true;
}

// EOF

```

Figure 195: Control Structures – if, elseif, else – Correct

▲ Control Structures

14.2. SWITCH, CASE

A switch structure looks like the following. Note the placement of parentheses, spaces, and braces.

- The case statement MUST be indented
- Separate each case logic with a blank line
- The case body must be indented twice
- The break keyword (or other terminating keywords) MUST be indented twice
- There MUST be a comment such as `// no break` when fall-through is intentional in a non-empty case body.

Example:

```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;
    :
        echo 'Default case';
        break;
}

```

Figure 196: Control Structures – switch, case – Example

✗ **Incorrect:** since `case 0` through `default` has not been indented once.

```
1.  <?php
2.
3.  switch ($expr) {
4.      case 0:
5.          echo 'First case, with a break';
6.          break;
7.
8.      case 1:
9.          echo 'Second case, which falls through';
10.         // no break
11.     case 2:
12.     case 3:
13.     case 4:
14.         echo 'Third case, return instead of break';
15.         return;
16.
17. default:
18.         echo 'Default case';
19.         break;
20. }
21.
22. // EOF
23.
```

Figure 197: Control Structures – switch, case – Incorrect

✗ **Incorrect**: since there are not two indents for `echo`, `break`, and `return`.

```
1.  <?php
2.
3.  switch ($expr) {
4.          case 0:
5.          echo 'First case, with a break';
6.          break;
7.
8.          case 1:
9.          echo 'Second case, which falls through';
10.         // no break
11.         case 2:
12.         case 3:
13.         case 4:
14.         echo 'Third case, return instead of break';
15.         return;
16.
17.         default:
18.         echo 'Default case';
19.         break;
20. }
21.
22. // EOF
23.
```

Figure 198: Control Structures – switch, case – Incorrect

✗ **Incorrect**: since `case 0`, `case 1` through `4`, plus `default`, are not separated by one blank line.

```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;
}

// EOF

```

Figure 199: Control Structures – switch, case – Incorrect

✓ **Correct**:

```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;
}

// EOF

```

Figure 200: Control Structures – switch, case – Incorrect

▲ Control Structures

14.3. WHILE, DO WHILE

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

✓ **Correct**:

```
1.  php
2.
3.  while ($expr) {
4.          // structure body
5.  }
6.
7.  do {
8.          // structure body;
9.  } while ($expr);
10.
11. // EOF
12.
```

Figure 201: Control Structures – while, do while – Correct

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

➕ **Example**:

```
9.<?php
10.
11. do {
12.     // structure body;
13. } while ($expr);
14.
15.  // EOF
16.
```

Figure 202: Control Structures – do while–Example

▲ Control Structures

14.4. FOR, FOREACH

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

Example:

```
1.  <?php
2.
3.  for ($i = 0; $i < 10; $i++) {
4.          // for body
5.  }
6.
7.  // EOF
8.
```

Figure 203: Control Structures – for Statement – Example

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

Example:

```
1.<?php
2.
3.foreach ($iterable as $key => $value) {
4.          // foreach body
5.  }
6.
7.  // EOF
8.
```

Figure 204: Control Structures – for each Statement – Example

Correct:

```
1.  <?php
2.
3.  for ($i = 0; $i < 10; $i++) {
4.          // for body
5.  }
6.
7.  foreach ($iterable as $key => $value) {
8.          // foreach body
9.  }
10.
11. // EOF
12.
```

Figure 205: Control Structures – for, foreach: – Correct

▲ Control Structures

14.5. TRY, CATCH, FINALLY

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

Example:

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

 // EOF

```

Figure 206: Control Structures – try, catch, finally – Example

Incorrect: since `catch` is not between `}` and `{` on a single line.

```php
php

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

// EOF

```

Figure 207: Control Structures – try, catch – incorrect

✓ **Correct**:

```php
<?php

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

// EOF

```

Figure 208: Control Structures – try, catch – Correct

▲ Control Structures

14.6. Alternative Syntax for Control Structures

PHP offers an alternative syntax for some of its control structures, namely, `if`, `while`, `for`, `foreach`, and `switch`. In each case, the basic form of the alternate syntax is to change the opening brace to a colon `:` and the closing brace to `endif:`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;` respectively.

✓ **Correct:**

```
1.  <?php if ($a == 5): ?>
2.  A is equal to 5
3.  <?php endif; ?>
4.
5.  // EOF
6.
```

Figure 209: Alternative Syntax –Correct

In the above example, the HTML block `"A is equal to 5"` is nested within an if statement and is written in the alternative syntax. The HTML block would be displayed only if `$a` is equal to `5`.

The alternative syntax applies to else and elseif as well. The following is an `if` structure with `elseif` and `else` in the alternative format:

✓ **Correct:**

```
1.  <?php
2.  if ($a == 5):
3.      echo 'a equals 5';
4.      echo '...';
5.  elseif ($a == 6):
6.      echo 'a equals 6';
7.      echo '!!!';
8.  else:
9.      echo 'a is neither 5 nor 6';
10. endif;
11.
12. // EOF
13.
```

Figure 210: Alternative Syntax – Correct

✗ **Incorrect:** Any output (including whitespace) between a switch statement and the first case will result in a syntax error. For example, this is invalid:

```
1. <?php switch ($foo): ?>
2. <?php case 1: ?>
3.
4. <?php endswitch ?>
5.
```

Figure 211: Alternative Syntax –Incorrect

Whereas this is valid, as the trailing newline after the switch statement is considered part of the closing `?>` and hence nothing is output between the switch and case:

Example:

```
1.  <?php switch ($foo): ?>
2.  <?php case 1: ?>
3.  …
4.  <?php endswitch ?>
5.
```

Figure 212: Alternative Syntax –Incorrect

▲ Control Structures

15. Declare Statements, Namespace, and Import Statements

A PHP file may contain a number of different blocks in its header. Whenever the following blocks below are present, they MUST be separated by a blank line and MUST NOT contain a blank line. Blocks MUST be in the order listed below, although non-relevant blocks can 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

If a file contains both HTML and PHP, any of the above sections can be used. In that case, they MUST appear at the top of the file, regardless of whether the remaining code consists of a closing PHP tag followed by 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.

An import statement MUST NEVER begin with a leading backslash since they MUST always be fully qualified.

Following is an example that illustrates a complete list of all possible blocks:

Example:

```php
<?php

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

declare(strict_types = 1);

namespace MyFlorida\License;

use MyFlorida\License\
{
ClassA as A, ClassB, ClassC as C;
};
use MyFlorida \License\SomeNamespace\ClassD as D;
use MyFlorida \License\AnotherNamespace\ClassE as E;

use function MyFlorida \License\
{
    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.
 */
class FooBar
{
    // ... additional PHP code ...
}

// EOF
```

Figure 213: Declare Statements, Namespace, and Import Statements – Example

A compound namespace with more than two depth levels MUST NOT be used. As a result, the following is the maximum depth of compounding: that can be used.

Example:

```php
<?php

use MyFlorida\License\SomeNamespace\
{
  SubnamespaceOne\ClassA,
  SubnamespaceOne\ClassB,
  SubnamespaceTwo\ClassY,
  ClassZ,
};

 // EOF

```

Figure 214: Compound Namespaces – Example

Additionally, the following will NOT be permitted:

Example:

```php
<?php

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

// EOF

```

Figure 215: Declare Statements, Namespace, and Import Statements – Example

▲

When wishing to declare strict types in files that contain markup outside of the PHP opening and closing tags, the declaration MUST be placed on the first line of the file and include the PHP opening tag, followed by the strict type declaration and the closing tag.

Example:

```
1.  <?php declare(strict_types = 1) ?>
2.  <html>
3.  <body>
4.    <?php
5.        // ... additional PHP code ...
6.    ?>
7.  </body>
8.  </html>
9.
```

Figure 216: Declare Statements, Namespace, and Import Statements – Declare Strict Types – Example

A declare statement must not contain any spaces and MUST be exactly `declare(strict_types=1)` (with an optional semi-colon terminator).

Block declare statements are permitted and MUST be formatted as shown in the following example. Take note of the placements of braces and spacing:

Example:

```
1.   declare(ticks = 1)
2.   {
3.
4.       // some code
5.   }
6.
```

Figure 217: Declare Statements, Namespace, and Import Statements – Declare Statements – Example

▲

16. Arrays

The PHP array is an ordered map that assigns values to keys. It can be used in several ways, including an array, list (vector), a hash table (an implementation of a map), dictionary, collection, stack, queue, and possibly more. Array values can be other arrays, trees, and multidimensional arrays are also possible.

Assignments the `=>` operator in arrays may be aligned with tabs. When splitting array definitions onto several lines, the last value may also have a trailing comma. This is valid PHP syntax and helps to keep code diffs minimal.

The array() construct can be used to create arrays. As arguments, it takes any number of comma-separated key `=>` value pairs.

Example:

```php
<?php

array(
    key  => value,
    key2 => value2,
    key3 => value3,
    ...
)

// EOF
```

Figure 218: Array – Syntax – Example

For multi-line arrays, the trailing comma is commonly used as it facilitates the addition of new elements at the end.

NOTE:

A short array syntax exists which replaces `array()` with `[]`.

Arrays can be created in two different ways. One way to specify the elements is to use array() to specify them as key-value pairs. A second method is to place all elements inside `[]`. When you create associate arrays with key pairs, there are two important things to keep in mind.

First, the key always has to be unique. PHP ignores all other key-value pairs in the array except the last one if you try to use the same key multiple times. Second, if a key is created using a float, boolean, or string representation of an integer, it will be cast to an integer.

Example: a simple array:

```php
<?php
$array = array(
    'foo' => 'bar',
    'bar' => 'foo',
);

// Using the short array syntax
    $array = [
    'foo' => 'bar',
    'bar' => 'foo',
];

// EOF

```

Figure 219: Simple Array – Example

The key can either be an int or a string. The value can be of any type.

▲

17. Classes

The concept of a class refers to a collection of variables and functions that work together in order to accomplish one or more specific tasks.

This section describes class files in terms of their names, definitions, properties, methods, and how to instantiate them.

1. Class File MUST only contain one definition and MUST be prefixed with class-.
   - i.e., `class User` → `class-user.php`, `class Office` → `class-office.php`
2. Class Namespace MUST be defined and MUST include vendor name.
   - e.g., `Namespace MyFlorida\DivTel;`, `Namespace MyFlorida\DivTel;`, `Namespace MyFlorida\Controller;`
3. Class Name MUST start with a capital letter and MUST be upper Camel-Case.
   - e.g., `MyFlorida`
4. Class Documentation MUST be present.
   - i.e., `@author`, `@global`, `@package`
5. Class Definition MUST place curly braces on their own line.
   - i.e., `class User ↵ { ↵ .. ↵ }`
6. Class Properties
   - MUST follow variable standards
   - MUST specify visibility
   - MUST NOT be prefixed with an underscore if private or protected
     - e.g., `$var1;`, `private $var2;`, `protected $var3;`
7. Class Methods
   - MUST follow function standards
   - MUST specify visibility
   - MUST NOT be prefixed with an underscore if private or protected

- e.g., `func1()`, `private func2()`, `protected func3()`

8. Class Instance

- MUST start with a capital letter
- MUST be Camel-Case
- MUST include parenthesis
  - e.g., `$user = new User();`, `$AccessFlorida = new AccessFlorida();`

▲

17.1. Class File

The Class files MUST contain only one definition and MUST include the prefix `class-`.

Filename: class-divtel.php.

✕ **Incorrect:** because `DivTelProgram` and `DivTel` should be defined in one file.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  Class DivTelProgram
6.  {
7.          // ...
8.  }
9.
10. Class DivTel
11. {
12.         // ...
13. }
14.
15. // EOF
16.
```

Figure 220: Class File – Incorrect

Filename: divtel.php.

✕ **Incorrect:** since there is no `class` prefix on the filename.

```
1.<?php
2.
3.  namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
2.  // ...
7. }
8.
9.  // EOF
10.
```

Figure 221: Class File – Incorrect

Filename: class-divtel.php

✓ **Correct:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.          // ...
8.  }
9.
10. // EOF
11.
```

Figure 222: Class File – Correct

Filename: class-csab.php

✓ **Correct:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class CSAB
6.  {
8.          // ...
9.  }
9.
10.  // EOF
11.
```

Figure 223: Class File – Correct

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.2. Class Namespace

Generally, all classes in PHP code reside in a namespace, except for core PHP classes. Core PHP classes do not reside in a namespace, which means they reside in the "root" namespace - much like a file at the root of your filesystem.

There MUST be a class namespace defined, and a vendor name included.

✗ **Incorrect**: because the namespace is not defined.

```
1.  <?php
2.
3.  class DivTelProgram
4.  {
5.          // ...
6.  }
7.
8.  // EOF
9.
```

Figure 224: Class Namespace – Incorrect

✗ **Incorrect**: since the vendor name is omitted from the namespace name.

```
1.  <?php
2.
3.  Namespace DivTel;
4.
5.  class DivTelProgram
6.  {
7.  // ...
8.  }
9.
10.  // EOF
11.
```

Figure 225: Class Namespace – Incorrect

✓ **Correct**:

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
11.         // ...
7.  }
8.
12. // EOF
9.
```

Figure 226: Class Namespace – Correct

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.3. Class Name

The class name MUST begin with a capital letter and MUST be written in upper Camel-Case, and the names must be nouns, never adjectives. Whenever you specify class names to PHP, be sure to reference the global namespace.

> **NOTE**:
>
> When specifying class names, a leading backslash is always used to reference the global namespace inside of namespaced code. When referencing a class name inside a string (e.g. given to the get-Method of the ObjectManager, in pointcut expressions or in YAML files), never use a leading backslash. It follows the notion of fully qualified names in strings that is native to PHP.

✗ **Incorrect:** since `DivTelProgram` doesn't begin with a capital letter.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class divtelProgram
6.  {
7.// ...
8.  }
9.
10. // EOF
11.
```

Figure 227: Class Namespace – Incorrect

✗ **Incorrect:** since `DivTelProgram` does not use upper Camel-Case.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class divtelProgram
6.  {
7.  // ...
8.  }
9.
10. // EOF
11.
```

Figure 228: Class Namespace – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.  // ...
8.  }
9.
9.// EOF
10.
```

Figure 229: Class Namespace – Correct

See Also: Class Names

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.4. Class Documentation

Documentation Blocks should precede hooks, actions, functions, methods, or class lines. To prevent the parser from becoming confused, the Doc-Block and declarations should not be separated by any opening/closing tags or other things.

1. Classes have their own documentation block describing the class's purpose.
2. There MUST be documentation for the class.
3. Additional tags or annotations, such as the following, should be added as needed.

+ Example:

```
1.  /**
2.  * Description including a code sample:
3.  *
4.  *       Namespace MyFlorida\DivTel;
5.  *       {
6.  *       class DivTelProgram
7.  *       public $var1;
8.  *       protected $_var2;
9.  *       private $_var3;
10. *
11. * First sentence is short description. Then you can write more, just as you like
12. *
13. * Here may follow some detailed description about what the class is for.
14. *
15. * Paragraphs are separated by an empty line.
16. /**
17.
```

Figure 230: Class Documentation – Annotation Tags –Example

✕ **Incorrect:** because `DivTelProgram` lacks documentation.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.          // ...
8.  }
9.
10. // EOF
10.
```

Figure 231: Class Documentation – Annotation Tags –Incorrect

✗ **Incorrect:** because there are no documentation tags for `user`.

```php
<?php

Namespace MyFlorida\DivTel;

/**
* DivTelProgram
*/
class DivTelProgram
{
        // ...
}

// EOF

```

Figure 232: Class Documentation – Incorrect

✓ **Correct:**

```php
<?php

Namespace MyFlorida\DivTel;

/**
* DivTelProgram
*
* @author:Firstname Lastname
* @global:object $post
* @Date: 05/30/2020
* @Category: MyFlorida
* @Link: https://www.phpdoc.org/
*/
class User
{
        // ...
}

// EOF

```

Figure 233: Class Documentation – Correct

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.5. Class Definition

In general, PHP classes, and more specifically, object-oriented programming, provide additional approaches to reusability, which can be applied to a wide variety of purposes.

- A class can describe entities with known properties and behaviors
- Functions and other objects can use a class for messages
- Related behaviors or states can be grouped together using a class
- A class provides a way to describe hierarchies and inheritance within an application

It is a blueprint, defining a combination of properties and behavior that serves as a template for objects or instances of the class.

Curly braces MUST be placed on their own line in class definitions.

✗ **Incorrect:** since { is not on its own line.

```php
<?php

Namespace MyFlorida\DivTel;

class DivTelProgram {
        // ...
}

// EOF

```

Figure 234: Class Definition – Incorrect

✓ **Correct:**

```php
<?php

Namespace MyFlorida\DivTel;

class DivTelProgram
{
        // ...
}

// EOF

```

Figure 235: Class Definition – Correct

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.6. Class Properties

1. Variables MUST be followed
2. Visibility MUST be specified
3. Private or protected MUST NOT be prefixed with an underscore

✗ **Incorrect:** because visibility for `$var1`, `$var2` and `$var3` has not been specified.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.          // Public
8.          $var1;
9.
10.         // Protected
11.         $var2;
12.
13.         // Private
14.         $var3;
15. }
16.
17. // EOF
18.
```

Figure 236: Class Properties – Incorrect

✗ **Incorrect:** since `_` is prefixed to `protected` and `private` properties.

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.        public $var1;
8.        protected $_var2;
9.        private $_var3;
10. }
11.
12. // EOF
13.
```

Figure 237: Class Properties – Incorrect

✓ **Correct:**

```
1.  <?php
2.
3.  Namespace MyFlorida\DivTel;
4.
5.  class DivTelProgram
6.  {
7.          public $var1;
8.          protected $var2;
9.          private $var3;
10. }
11.
12. // EOF
13.
```

Figure 238: Class Properties – Correct

✚ **Example**: Class and Properties. At its most basic, classes syntactically are only a form of declaration:

```
1.Class MotorVehicles
2.{
3.}
4.
```

Figure 239: Class and Properties – Example

There are many kinds of motor vehicles: racecars, automobiles, motorcycles, trucks, riding lawnmowers, lorries, and more. These are distinguished from one another by a variety of properties and behaviors.

We can now define its number of wheels `$wheelCount`, what type of fuel `$fuelType`, the engine horse power `$engineHP`, and its weight `$weight`. Is it street legal? `$streetLegal`. With each, a valid type is defined so that we always know what we'll receive from the class property, as well as what values will be accepted

We can expand our `MotorVehicles` class by adding some properties:

Example:

```
1.{
2.    int $wheelCount;
3.    bool $usesFuel;
4.    bool $selfPropelled;
5.    float $weight;
6.}
7.
```

Figure 240: Class Properties – Example

Example:

```
1.  <?php
2.  Class MotorVehicles
3.  {
4.       int $wheelCount;
5.       string $fuelType;
6.       float $weight;
7.       int $engineHP;
8.       bool $streetLegal
9.  }
10.
11. // EOF
12.
```

Figure 241: Class Properties – Example

Then, we can create an instance of a motor vehicle using the `new` keyword:

Example:

```
1.  $automobile = new WheeledVehicle();
2.
```

Figure 242: Class Instance – Example

From here, we can better define our `$automobile` instance:

Example:

```
1.  $ MotorVehicle ->wheelCount    = 6;
2.  $ MotorVehicle ->fuelType      = diesel;
3.  $ MotorVehicle ->weight        = 2200;
4.  $ MotorVehicle ->engineHP      = 427;
5.  $ MotorVehicle ->streetLegal   = true
6.
```

Figure 243 Class Properties – Example

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.7. Class Methods

Methods are functions that are specific to a class instance. It means that the class instance can access the values assigned to its properties; in the case of the example above, the `$automobile` is an instance of the class `MotorVehicles`, so we can access the values that we have based on the methods we assigned.

Here is a method that compares the number of wheels in an instance with that of another instance; using the method, we could sort `WheeledVehicles` instances according to the number of wheels each has.

Inside our Motor Vehicles class, let's define a method.

Example:

```
1.  Class MotorVehicle
2.   {
3.     int $wheelCount;
4.     int $engineHP
5.     string  $fuelType;
6.     bool $streetLegal;
7.     float $weight;
8.
9.     public function compareWheelCount(MotorVehicles $comparison): int
10.      {
11.          return $this->wheelCount <=> $comparison->wheelCount;
12.     }
13. }
```

Figure 244 Class Methods – Example

If we had several MotorVehical instances, we could sort an array of them as follows:

Example:

```
1.  usort($MotorVehicles, fn($a, $b) => $a->compareWheelCount($b));
2.  //*Starting in PHP 7.4, you can define 'arrow functions', which are a compact way of
3.  * defining closures that return the result of exactly one expression. The above could
4.  * also be written:
5.  */
6.  usort($motorVehicles, function ($a, $b) {
7.     return $a->compareWheelCount($b)
8.});
9.
```

Figure 245 Class Properties –Example

Class methods:

1. Visibility MUST be declared on all methods
2. MUST adhere to functions standards
3. Visibility MUST be declared on all methods
4. 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
5. A space MUST NOT be added after a method or function name. The opening brace MUST be on its own line, and the closing brace MUST be placed on the next line following the body. After the opening parenthesis and the closing parenthesis MUST NOT be preceded by a space.

The following is an example of a method declaration. Note the placement of parentheses, commas, spaces, and braces:

Example:

```php
<?php

namespace MyFlorida\DivTel;

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

```

Figure 246 Class Methods – Method Declaration

✗ **Incorrect:** since visibility is not specified for `get_var1()`, `get_var2()` and `get_var3()`.

```php
<?php

Namespace MyFlorida\DivTel;

class User
{
        // ...

        // Public
        function get_var1() {
                return $this->var1;
        }

        // Protected
        function get_var2() {
                  return $this->var2;
}

        // Private
        function get_var3() {
                  return $this->var3;
    }
}

// EOF

```

Figure 247: Class Methods – Incorrect

✗ **Incorrect:** since `protected` and `private` methods have a prefix of `_`.

```php
<?php

Namespace MyFlorida\DivTel;

class User
{
        // ...

        public function get_var1() {
                return $this->var1;
        }

        protected function _get_var2() {
                return $this->var2;
        }

        private function _get_var3() {
                return $this->var3;
        }
}

// EOF

```

Figure 248: Class Methods – Incorrect

✓ **Correct:**

```php
<?php

namespace MyFlorida\DivTel;

class User
{
        // ...

        public function get_var1() {
                return $this->var1;
        }

        protected function get_var2() {
                return $this->var2;
        }

        private function get_var3() {
                return $this->var3;
        }
}

// EOF

```

Figure 249: Class Methods – Correct

See Also: Method Names

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.8. Class Instance

A class is a blueprint that is used to create objects. The term object refers to something that is an instance of a class - it's the concrete 'thing' that is made using a specific class. In other words, 'object' and 'instance' are the same thing, but 'instance' refers to the relationship between an object and the class it belongs to.

The instance variable reserves memory for the data the class needs. We will assume that you would like to add a place for a string or int variable. An instance variable can be used to reserve memory for the object's lifetime. In this way, each variable in the object will have its own unique memory.

Class Instance:

1. Variables MUST be followed
2. Parenthesis MUST be included

✗ **Incorrect**: since parenthesis are missing in the `new AccessFlorida`.

```
1.  <?php
2.
3.  $DivTelProgram = new SUNCOMProgram;
4.
5.  // EOF
6.
```

Figure 250: Class Instance – Incorrect

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.9. PHP INHERITANCE

As a result of a class deriving from another class, the child class will also inherit all of the public and protected properties and methods from the parent class. In addition, it can also establish its own properties and methods. The `extends` keyword is used in order to define an inherited class.

It would be nice not to have to set all the properties of a car. What if we could define those and still have the concept of a "motor vehicle"?

This is possible through class extensions. In this example, we are going to create a class `Automobile` that extends `MotorVehicles`, and define some of those properties up front:

Example:

```
1.{
2.    int $wheelCount     = 4;
3.    bool $usesFuel      = true;
4.    bool $selfPropelled = true;
5.}
6.
```

Figure 251: Class Inheritance – Example

It is now possible to define our car in the following manner:

Example:

```
1.$sedan = new Car();
2.$sedan->weight = 2200;
3.
```

Figure 252: Class Inheritance – Example

If we were to access the other properties, we'd find they have same defaults set:

Example:

```
1.if ($sedan->usesFuel && $sedan->selfPropelled) {
2.    printf("Vehicle has %d wheels.\n", $sedan->wheelCount); // "Vehicle has 4 wheels"
3.}
4.
```

Figure 253 Class Inheritance – Example

The extends keyword tells PHP that the new class we are declaring will inherit from the class specified. This means that, unless we override them, the new class will have the same properties and behaviors.

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

17.10. Class Constructor Calls

Constructors are methods called when a new instance is created, and they can accept arguments but must not define return values. In most cases, constructors are used to initialize properties based on the arguments they receive. PHP constructors must be named `__construct()`. The `__` prefix signifies a reserved "magic" method used by the language for certain behaviors.

When calling class constructors without arguments, always include parentheses:

Example:

```
1.  $foo = new MyClassName();
2.
```

Figure 254: Class Constructor Calls – No Arguments –Example

This is to maintain consistency with constructors that have arguments:

Example:

```
1.  $foo = new MyClassName($arg1,
2.  arg2);
3.
```

Figure 255: Class Constructor Calls – Example Consistency with Arguments –Example

If the class name is a variable, the variable will first be evaluated to get the class name, and then the constructor will be called. Use the same syntax:

Example:

```
1.   $bar = 'MyClassName';
2.   $foo = new $bar();
3.   $foo = new $bar($arg1, $arg2);
```

Figure 256: Class Constructor Calls – Constructor –Example

We can write a constructor that assigns a `$weight` value based on the class instance `Automobile` that we defined earlier.

Example:

```
1.class Automobile extends WheeledVehicles
2.{
3.    int $wheelCount     = 4;
4.    bool $usesFuel      = true;
5.    bool $selfPropelled = true;
6.
7.    public function __construct(float $weight)
8.    {
9.        $this->weight = $weight;
10.     }
11. }
12.
```

Figure 257: Class Constructor Calls – Constructor –Example

The constructor accepts a $weight argument and then assigns it to a property. But we didn't define that property! Why? Because it's defined already in the parent class, `MotorVehicles`.

▲ – Example

The key can either be an int or a string. The value can be of any type.

▲

Classes

18. PHP Error Reporting

To provide an overview of the error reporting process, this section explains how it works, how the information can be gathered, and how it can be interpreted.

It may be necessary to edit the php.ini file to provide the full range of error reporting options.

In most cases, adding these lines to the top of a php file will provide some of the error information without editing the `php.ini` file.

Example:

```
1.  ini_set('display_errors', 1);
2.  ini_set('display_startup_errors', 1);
3.  error_reporting(E_ALL);
4.
```

Figure 258: Error Reporting – E_ALL –Example

Parsing the code this way won't produce errors like improper formatting. A blank page would result since the script is parsed prior to execution.

The following line must be added `to` `php.ini` in order to see these errors:

Example:

```
1.  display_errors = on
2.
```

Figure 259: Error Reporting – Display Errors

NOTE:

For a production server, it is important to note that this function is not left on!

Another option is to edit the `.htaccess configuration file`. This way, it may avoid accidentally carrying over the setting to the production server.

Add the following lines:

Example:

```
1.  php_flag display_errors on
2.  php_flag display_startup_errors on
3.
```

Figure 260: Error Reporting – Edit .htaccess –Example

From here, errors can be output to a log file, providing feedback even when a page fails to load.

Add the following line.

Example:

```
1.  php_value error_log log/errors_php.log
2.
```

Figure 261: : Error Reporting – Edit .htaccess –Example

For PHP 5.4 and up, the default error reporting value is:

Example:

```
1.E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED
2.
```

Figure 262: Error Reporting – Default Error Reporting Value –Example

Thus, all errors are displayed by default except `E_NOTICE`, `E_STRICT`, and `E_DEPRECATED`.

By replacing the values with your chosen levels, separated by an inclusive OR, you can also specify the error levels.

Example:

```
1.error_reporting(E_WARNING | E_PARSE);
2.
```

Figure 263: Error Reporting – E_PARSE –Example

Additionally, use `AND NOT` to show only certain types of errors.

Using this line, all errors except non-fatal runtime warnings will be reported:

Example:

```
1.error_reporting(E_ALL & ~E_WARNING);
2.
```

Figure 264: Error Reporting –E_ALL, and E_WARNING –Example

▲ PHP Error Reporting

18.1. Predefined Constants and Bitmask Values in PHP7

Below is a list of all of the options for specifying which errors to report. Not that the constant names will only work inside of php.ini or in a php file- if you are adding these to your .htaccess file, you must use the bitmask value.

This section lists all the options for specifying which errors to report. The constant names only work inside of `php.ini` or in a php file. If you're adding them to .htaccess, the bitmask must be used.

▲ PHP Error Reporting

Constant	Value	Meaning
E_ERROR	1 (int)	A fatal runtime error. These errors are not recoverable, such as memory allocation errors.
E_WARNING	2 (int)	In most cases, the errors you may see are warnings, and they are non-fatal errors, and execution of the script is not halted.
E_PARSE	4 (int)	If there's a mistake in the syntax of the code it's trying to parse, this error will occur.
E_NOTICE	8 (int)	This is less of a warning but something that merits attention nonetheless. Often you will see these when all is fine, but sometimes they manage to catch a serious issue before it has had time to permeate.
E_CORE_ERROR	16 (int)	The PHP core caused a fatal runtime crash during the initial startup.
E_CORE_WARNING	32 (int)	PHP core generated a fatal error at startup.
E_COMPILE_ERROR	64 (int)	What happens when a fatal error occurs during the compilation of the Zend Scripting Engine.
E_COMPILE_WARNING	128 (int)	It is similar to an E_WARNING, except the Zend framework generates it.
E_USER_ERROR	256 (int)	Using trigger_error() string, the developer intentionally triggers a fatal error.
E_USER_WARNING	512	Warning message generated by the user. It is similar to an E_WARNING, but it is generated by PHP code using trigger_error().
E_USER_NOTICE	1024 (int)	The PHP function trigger_error() is used to generate this error message, which is similar to the E_NOTICE.

Constant	Value	Meaning
E_STRICT	2048 (int)	PHP warning when code may become deprecated and/or non-functional in the future. It is not an actual error and is no longer very useful in PHP 7.
E_RECOVERABLE_ERROR	4096 (int)	Application aborts as an E_ERROR if a user-defined error handler does not catch it.
E_DEPRECATED	8192 (int)	This runtime notice informs that while a specific piece of code is working right now, it will not be supported in future versions.
E_USER_DEPRECATED	16384 (int)	Similar to an E_DEPRECATED, but generated in PHP code using the trigger_error() function.
E_ALL	32676 (int)	All errors, warnings, and notices are supported.

Table 6: Predefined Constants and Bitmask Values

The following lines of code display all the flags that are set. A blank line indicates the absence of a flag.

Example:

```php
<?php
$errLvl = error_reporting();
For ($i = 0; $i < 15;  $i++ ) {
print FriendlyErrorType($errLvl & pow(2, $i)) . '<br>\\n';
}

function FriendlyErrorType($type)
{
switch($type)
{
case E_ERROR: // 1 //
        return 'E_ERROR';
case E_WARNING: // 2 //
        return 'E_WARNING';
case E_PARSE: // 4 //
        return 'E_PARSE';
case E_NOTICE: // 8 //
        return 'E_NOTICE';
case E_CORE_ERROR: // 16 //
        return 'E_CORE_ERROR';
case E_CORE_WARNING: // 32 //
        return 'E_CORE_WARNING';
case E_COMPILE_ERROR: // 64 //
        return 'E_COMPILE_ERROR';
case E_COMPILE_WARNING: // 128 //
        return 'E_COMPILE_WARNING';
case E_USER_ERROR: // 256 //
        return 'E_USER_ERROR';
case E_USER_WARNING: // 512 //
        return 'E_USER_WARNING';
case E_USER_NOTICE: // 1024 //
        return 'E_USER_NOTICE';
case E_STRICT: // 2048 //
        return 'E_STRICT';
case E_RECOVERABLE_ERROR: // 4096 //
        return 'E_RECOVERABLE_ERROR';
case E_DEPRECATED: // 8192 //
        return 'E_DEPRECATED';
case E_USER_DEPRECATED: // 16384 //
        return 'E_USER_DEPRECATED';
}
 return '';}

// EOF

```

Figure 265: Error Reporting – Predefined Constants

In order to exclude warnings from every flavor, you can do the following:

+ Example:

```
1.  error_reporting(E_ALL & ~E_WARNING & ~E_CORE_WARNING & ~E_COMPILE_WARNING &
    ~E_USER_WARNING);
2.
3.  //EOF
4.
```

Figure 266: Error Reporting – Predefined Constants

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

19. Best Practices

PHP best practices have always been to improve the developer experience. In general, before committing to a project, you have to fully understand all its benefits and drawbacks. The use of modern PHP development concepts imparts a true and easy way to write software.

1. Comments
2. Naming Conventions
3. Code Format
4. Single Quotes
5. Functions inside Loopsever Use Functions inside Loops
6. PHP Error Reportings
7. Using the DRY Approach
8. Variable Initialization
   - SHOULD be initialized prior to use and SHOULD occur early.
     - e.g., `$var1 = '';`, `$var2 = 0;`
9. Initialization/Declaration Order
   - MUST lead with globals, follow with constants, conclude with local variables
   - MUST lead with properties and follow with methods in classes
   - MUST lead with public, follow with protected, conclude with private methods in classes
   - SHOULD be alphabetical within their type
   - i.e., `global $var1;`, `define('VAR2', '');`, `$var3 = 0;`
10. Globals SHOULD NOT be used
    - i.e., no `global $var;`
11. Explicit Expressions SHOULD be used
    - e.g., `If ($expr === false), while ($expr !== true)`
12. PHP Error Reporting MUST NOT trigger errors

- i.e., do not use deprecated functions, etc.

13. Constants
14. PHP and regex
15. PHP and UTF-8
16. Working with dates and times

▲

19.1. FUNCTIONS INSIDE LOOPS

The value should be counted in advance because, if it is put it in the loop, the function will be executed every time the condition is met.

✗ **Incorrect:**

```
<?php
for ($i = 0, $i <= count($array);  $i++)

{
//statements
}
// EOF
```

Figure 267: Functions inside Loops – Incorrect

✓ **Correct:**

```
<?php

$count = count($array);

for ($i = 0; $i < $count;  $i++)
{
//statements
}

// EOF
```

Figure 268: Functions inside Loops – Correct

See Also: Function Names

See Also: Functions

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.2. DRY Approach

DRY or Don't Repeat Yourself. The concept of reducing code redundancy in software engineering tries to cut down redundancy in code.

This concept is not PHP-specific, but can be applied to any programming language such as Java, C++, etc.

✕ **Incorrect:**

```
1.        $mysql = mysql_connect('localhost', 'admin', 'admin_pass');
2.
3.        mysql_select_db('csab' or die('Cannot Select Database. ');
4.
```

Figure 269: DRY Approach – Incorrect

✓ **Correct**: using the DRY approach, the code will look something like this:

```
1.  $db_host  = 'localhost';
2.
3.  $db_user = 'admin';
4.
5.  $db_pass = 'admin_pass';
6.
7.  $db_name = 'csab'
8.
9.  $mysql = mysql($db_host, $db_user, $db_pass);
10.
11. mysql_select_db($db_name);
12.
```

Figure 270: DRY Approach – Correct

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.3. VARIABLE INITIALIZATION

Variable initialization refers to assigning an initial value to a variable before being used. In the absence of initialization, a variable would have an unknown value, leading to unpredictable results in calculations or other operations when the variable is used.

Declaring a variable should be followed by initializing it. Variable initialization can be explicit or implicit. If a value is assigned to variables in a declaration statement, they are explicitly initialized. If a value is assigned to variables during processing, they are implicitly initialized.

Initialization of variables SHOULD occur before use and SHOULD occur early.

✓ **Acceptable**: but `$movies` should be initialized prior to use.

```
1.  <?php
2.
3.  $movies = get_movies();
4.
5.  // EOF
6.
```

Figure 271: Variable Initialization – Acceptable

✓ **Acceptable**: although the `$movies` variable should be initialized earlier.

```
1.  <?php
2.
3.  if ($expr) {
4.          // ....
5.  }
6.
7.  $movies = array();
8.  $movies = get_movies();
9.
10. // EOF
11.
```

Figure 272: Variable Initialization – Acceptable

✓ **Preferred**:

```php
<?php

$movies = array();

if ($expr) {
// ....
}

$movies = get_movies();

 // EOF

```

Figure 273: Variable Initialization – Preferred

See Also: Variables

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.4. Initialization/Declaration Order

Order of initialization/declaration:

- Globals MUST come first, followed by constants, and conclude with the local variables
- Properties MUST come first, followed by methods in classes
- Public MUST be followed by protected, then by private methods in classes
- SHOULD be arranged alphabetically by type

✗ **Incorrect**: since `$app_config` is not the first item, `ENVIRONMENT` is not the second, and `$id` is not the third.

```php
<?php

define('ENVIRONMENT', 'PRODUCTION');

$id = 0;

global $app_config;

// EOF

```

Figure 274: Initialization/Declaration Order – Incorrect

✗ **Incorrect**: because `get_name()` should be declared before `$name`.

```php
<?php

Namespace MyFlorida\DivTel;

class Office
{
        public function get_name() {
                // ...
        }

        private $name;
}

// EOF

```

Figure 275: Initialization/Declaration Order – Incorrect

✗ Incorrect: because `get_id()` should be first, `get_status()` should be second, and `get_name()` should be third.

```php
<?php

Namespace MyFlorida\DivTel;

class Office
{
        private $id;
        private $name;
        private $status;

        private function get_name() {
                // ...
        }

        public function get_id() {
                // ...
        }

        protected function get_status() {
                // ...
        }
}

// EOF

```

Figure 276: Initialization/Declaration Order – Incorrect

✓ **Acceptable**: This is acceptable, but the globals and constants should be listed alphabetically.

```php
<?php

Global $db_connection,
$app_config,
$cache;

define('MODE', 1);
define('ENVIRONMENT', 'PRODUCTION');

$id = 0;
$firstname = '';
$lastname = '';

// EOF

```

Figure 277: Initialization/Declaration Order – Acceptable

✓ **Acceptable**: however, `get_actors` should be declared before `get_movies`.

```
1.  <?php
2.
3.  Function get_movies() {
4.          // ...
5.  }
6.
7.  `function get_actors() {
8.          // ...
9.}
10.
11. // EOF
12.
```

Figure 278: Initialization/Declaration Order – Acceptable

✓ **Correct**:

```
1.  <?php
2.
3.  global $app_config,
4.          $cache,
5.          $db_connection;
6.
7.  define('ENVIRONMENT', 'PRODUCTION');
8.  define('MODE', 1);
9.
10. $id = 0;
11. $firstname = '';
12. $lastname = '';
13.
14. // EOF
15.
```

Figure 279: Initialization/Declaration Order – Correct

✓ **Correct**:

```php
<?php

namespace MyFlorida\DivTel;

class Office
{
        private $id;
        private $name;
        private $status;

        public function get_id() {
                // ...
        }

        protected function get_status() {
                // ...
        }

        private function get_name() {
                // ...
        }
}

// EOF

```

Figure 280: Initialization/Declaration Order – Correct

✓ **Preferred**:

```php
<?php

function get_actors() {
        // ...
}

function get_movies() {
        // ...
}

// EOF

```

Figure 281: Initialization/Declaration Order – Preferred

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.5. GLOBALS

Use of globals SHOULD be avoided.

$GLOBALS is a PHP super global variable used to access global variables from anywhere in the PHP script (also from within functions or methods). PHP stores all global variables in an array called $GLOBALS[index]. The index holds the name of the variable.

Why not use global variables PHP?

The problem with non const global variables is that any function can alter them, making them a dangerous concept. The use of global variables reduces the modularity and flexibility of the program, allowing for testing and the ability to reuse it elsewhere. Rather than using global variables, there are numerous advantages to using local variables instead.

✓ **Acceptable**: but global variables need to be avoided.

```
1.  <?php
2.
3.  $pdo = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
4.
5.  function get_user($id) {
6.          global $pdo;
7.          // ...
8.  }
9.
10. // EOF
11.
```

Figure 282: Globals – Acceptable

✓ **Preferred**:

```php
<?php

function get_database_object() {
        return new PDO('mysql:host=localhost;dbname=test', $user, $pass);
}

function get_suser ($id) {
        $pdo = get_database_object();
        // ...
}

// EOF

```

Figure 283: Globals – Preferred

See Also: Global Variables

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.6. Explicit Expressions

Let's say you want to write a cookie library, and you wish to create a function to determine whether a cookie's value has already been set. Perhaps that function checks that $cookie equals `==` `null`. Another developer later decides to store a path in a cookie. There could be a page that calls the library function in order to check if a cookie is unset, and if it is, it defaults the path to `'/root'`. Maybe an empty string if a valid cookie value. Then, if an empty string is ever set to the cookie value, the cookie library will think the cookie was not set and default it to `'/root'`, rendering it impossible to persist an empty path as a cookie value. What disorder. The best way to avoid this nuisance is to be explicit when checking for equality.

It is not necessary to rely on double equals. Simply put, it would it is just being lazy.

✓ **Acceptable**: however, an alternative would be to use `===` instead.

```php
<?php

if ($expr == true) {
        // ...
}

// EOF

```

Figure 284: Explicit Expressions – Acceptable

✓ **Preferred**:

```php
<?php

if ($expr === true) {
// ...
}

// EOF

```

Figure 285: Explicit Expressions – Preferred

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.7. CONSTANTS

`define()` vs. `const`

Use `define()` unless readability, class constants, or micro-optimization are concerns.

The traditional way of defining constants in PHP is by using a function called `define()`. PHP gained the capability of declaring constants with the const keyword at some point. Which one should you use when defining your constants?

What differentiates the two methods lies in the slight differences between them.

`define()` defines constants at run time, while `const` defines constants at compile time. There is a very slight advantage to `const` here. But it is not a problem you need to worry about unless you are building large-scale applications.

`define()` includes your constants in the global scope, though you can also include the namespaces in the constant name. As a consequence, you cannot use `define()` to declare class constants.

The `define()` function allows you to use expressions both within the constant name and the constant value, as opposed to `const`, which does not allow either. This makes `define()` much more flexible.

`define()` can be called within an `if()` block, while const cannot.

Example:

```php
<?php
// Let's see how the two methods treat namespaces
namespace MiddleEarthCreatures\Dwarves;
const GIMLI_ID = 1;
define('MiddleEarth\Creatures\Elves\LEGOLAS_ID', 2);

print(\MiddleEarth\Creatures\Dwarves\GIMLI_ID); // 1
print(\MiddleEarth\Creatures\Elves\LEGOLAS_ID); // 2; note that we used define(), but the namespace is still recognized

// Now let's declare some bit-shifted constants representing ways to enter Mordor.
define('TRANSPORT_METHOD_SNEAKING', 1 << 0); // OK!
const TRANSPORT_METHOD_WALKING = 1 << 1; // Compile error! const can't use expressions as values

// Next, conditional constants.
define('HOBBITS_FRODO_ID', 1);

if($isGoingToMordor){
    define('TRANSPORT_METHOD', TRANSPORT_METHOD_SNEAKING); // OK!
    const PARTY_LEADER_ID = HOBBITS_FRODO_ID // Compile error: const can't be used in an if block
}

// Finally, class constants
class OneRing{
    const MELTING_POINT_CELSIUS = 1000000; // OK!
    define('MELTING_POINT_ELVISH_DEGREES', 200); // Compile error: can't use define() within a class
}
// EOF

```

Figure 286: define() vs. const – Example

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.8. PHP AND REGEX

Use the PCRE (preg_*) family of functions.

Until PHP 7 came along, there were two methods for specifying regular expressions in PHP: the PCRE (Perl-compatible, `preg_*`) functions and the POSIX (POSIX extended, `ereg_*`) functions.

The regular expression used in each of these families of functions varies slightly. Fortunately for us, the `ereg_*` functions have been removed in PHP 7, so this source of confusion is no longer an issue.

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.9. PHP AND UTF-8

UTF-8 in PHP sucks. Sorry.

PHP does not support Unicode at a low level at the moment. While there are ways to ensure that UTF-8 strings are processed correctly, it is not as simple as it seems.

The task requires digging through almost all of the levels of the web app, from HTML to SQL to PHP. We'll aim for a brief, practical summary.

UTF-8 at the PHP level

UTF-8 does not require anything special for basic string operations, such as concatenating two strings and assigning strings to variables. However, most string functions, such as `strpos()` and `strlen()`, do require some special consideration. It is often the case that these functions have an `mb_*` counterpart: for example, the function `mb_strpos()`, or the function `mb_strlen()`. Both of these counterpart functions are referred to as multibyte string functions when they are used together. These multibyte string functions are specially designed to be used to process Unicode strings.

19.10. WORKING WITH DATES AND TIMES

Use the `DateTime` class.

The only way to work with dates and times in PHP in the old days was to use a bewildering combination of `date()`, `gmdate()`, `date_timezone_set()`, `strtotime()`, and so on. Unfortunately, you can still find many tutorials on the internet that feature these old-fashioned and difficult functions.

For our benefit, recent versions of PHP include the much friendlier `DateTime` class. Using this class, you can access all the functionality and more of the old date functions in one simple class, making the time zone conversion process so much simpler than before. The `DateTime` class should always be used when creating, comparing, changing, and displaying dates in PHP.

+ Example:

```
1.  <?php
2.  // Construct a new UTC date.  Always specify UTC unless you really know what you're
    doing!
3.  $date = new DateTime('2011-05-04 05:00:00', new DateTimeZone('UTC'));
4.
5.  // Add ten days to our initial date
6.  $date->add(new DateInterval('P10D'));
7.
8.  print($date->format('Y-m-d h:i:s')); // 2011-05-14 05:00:00
9.
10. // Sadly we don't have a Middle Earth timezone
11. // Convert our UTC date to the PST (or PDT, depending) time zone
12. $date->setTimezone(new DateTimeZone('America/Los_Angeles'));
13.
14. // Note that if you run this line yourself, it might differ by an hour depending on
    daylight savings
15. print($date->format('Y-m-d h:i:s')); // 2011-05-13 10:00:00
16.
17. $later = new DateTime('2012-05-20', new DateTimeZone('UTC'));
18.
19. // Compare two dates
20. if($date < $later){
21.     print('Yup, you can compare dates using these easy operators!');
22. }
23.
24. // Find the difference between two dates
25. $difference = $date->diff($later);
26.
27. print('The 2nd date is ' . $difference->days . ' later than 1st date.');
28.
29. // EOF
30.
```

Figure 287: Dates and Times – Example

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

19.11. CHECKING IF A VALUE IS NULL OR FALSE

With PHP's loose typing system, it is possible to check the value of a variable in a variety of ways. On the other hand, it also presents a number of problems. The use of == to check if a value is null or false can result in false positives if the variable has an empty string or 0 as its value. There is a method called `istet()` that checks if a variable has a value that is not null, but it does not check against the boolean false.

There are two functions that check whether a value is null: `is_null()` and `is_bool()`, but there's one more that can be used: the `===` operator. `===` indicates that the values are identical. This is not the same as equivalent in PHP's loosely-typed world. This is not only faster than `is_null()` and `is_bool()`, but it is also more aesthetically pleasing than using a function for comparison.

+ Example:

```
1.  <?php
2.  $x = 0;
3.  $y = null;
4.
5.  // Is $x null?
6.  if($x == null){
7.      print('Oops! $x is 0, not null!');
8.  }
9.
10. // Is $y null?
11. if(is_null($y)){
12.     print('Great, but could be faster.');
13. }
14.
15. if($y === null){
16.     print('Perfect!');
17. }
18.
19. // Does the string abc contain the character a?
20. if(strpos('abc', 'a')){
21.     // GOTCHA!  strpos returns 0, indicating it wishes to return the position of the
    first character.
22.     // But PHP interpretes 0 as false, so we never reach this print statement!
23.     print('Found it!');
24. }
25.
26. //Solution: use !== (the opposite of ===) to see if strpos() returns 0, or boolean
    false.
27. if(strpos('abc', 'a') !== false){
28.     print('Found it for real this time!');
29. }
30. // EOF
31.
```

Figure 288: === operator

▲

It is this information that constitutes the essence of PHP error reporting.

▲ PHP Error Reporting

Best Practices

20. APPENDICES

20.1. TOOL CHAIN

Tool Name	Description
Aptana Studio	Open-source web development
CakePHP	Prototype, Validate
Code Sniffer	PHP_CodeSniffer tokenizes PHP files and detects violations of a defined set of coding standards
CodeIgniter	CodeIgniter
Eclipse Foundation	The PHP IDE project delivers a PHP Integrated Development Environment framework for the Eclipse platform
Composer	A Dependency Manager for PHP
Heredoc	Generate multi-line strings without messing up code indentation
Laminas	enterprise-ready PHP Framework and components
Local-PHP-Security-Checker	Local PHP Security Checker is a command-line tool that checks if a PHP application depends on PHP packages with known security vulnerabilities. It uses the Security Advisories Database behind the scenes
Modern-PHP-Cheatsheet	This document is a cheatsheet for PHP you will frequently encounter in modern projects, and most contemporary sample code

Tool Name	Description
NowDoc	Provides an alternative way of defining strings in PHP
PhpStorm	PhpStorm is a cross-platform IDE
Open Web Application Security Project	Open Web Application Security Project® (OWASP) is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web
PHP Debug Bar	Integrates easily in any project and can display profiling data from any part of an application. It is built with data collectors for standard PHP feature
PHP Coding Standards Fixer	Fixes code to follow standards, whether PHP coding standards as defined in the PSR-1, PSR-2, etc. or other community-driven like Symfony
phpDocumentor	phpDocumentor is the de-facto documentation application for PHP projects
PHPUnit	PHPUnit is a programmer-oriented testing framework for PHP
PHP Website	Fast, flexible, and pragmatic, PHP powers everything suited to web development

Tool Name	Description
Selenium	Automates browsers. Primarily it is for automating web applications for testing purposes
Survive The Deep End: PHP Security	Trust nobody and nothing Assume a worst-case scenario
Symfony	Symfony is a set of reusable PHP components
Zend Framework	Performance tuning
Xdebug 3	

Table 7: PHP Tool Chain

▲

21. Table of Figures

▲

22. Table of Tables

▲

23. References

23.1. Fair Use

The Copyright Law of the United States provides legal protection for intellectual property.

The fair use of a copyrighted work, including reproduction through copies or phonorecords, or by other means specified in Section 106, for criticism, commentary, news reporting, teaching, scholarship, or research, is not an infringement of the copyright.

The preceding paper is derived from research, content, code blocks, examples, and paraphrasing of references cited and embedded in the Reference section of this document, as defined by the "Fair Use" rule of copyright, 17 U.S. Code, §106A.

▲

23.2. PUBLICATIONS

Ankitha, and Samvada: (2019, December 24). *An Introduction to PHP Standard Recommendation (PSR)*.

An Introduction to PHP Standard Recommendation (PSR) | Specbee — Retrieved from "https://www.specbee.com/blogs/introduction-php-standard-recommendation-psr."

Specbee creates content management solutions specializing in Drupal development.

Publications are licensed under GPL-2.0-or-later.

Cabal, Alex. PHP Best Practices: (26 July 2021). *A Short, Practical Guide for Common and Confusing PHP Tasks*.

PHP Best Practices — Retrieved from "https://phpbestpractices.org."

PHP Best Practices by Alex Cabal is dedicated to the public domain via the CC0 1.0 Universal Public Domain via the CC0 1.0 Universal Public Domain Dedication and is thus free of copyright restrictions worldwide.

Cullum, M., Szanto, K., Makarov, A., and the PHP *Fr*amework Interoperability *Group:* (2019, August 9). *PSR-12: Extended Coding Style - PHP-FIG. PSR-12*.

Extended Coding Style — Retrieved from "https://www.php-fig.org/psr/psr-12".

The PHP Standard Recommendation (PSR) extends and expands on PSR-1, the basic coding standard. The aim is to enable the interoperability of components and provide a common technical basis for implementing proven concepts for optimal programming and testing practices.

Flow Framework: © (2006 and onwards) by the authors, *The Definitive Guide — Flow Framework* dev-*master documentation*.

The Definitive Guide — Retrieved from "https://flowframework.readthedocs.io/en/stable/StyleGuide/index.html."

Flow is a free PHP framework licensed under the MIT license.

The Publications Style Guide provides editorial guidelines for text in all kinds of publications, technical documentation, and the software user interface of applications issued by the Neos Project. Anybody writing text is encouraged to use this document as a guide to writing style, usage, and specific terminology.

Code Grepper: (2020) Grepper Technologies LLC, Grepper Dev Community, Hawkins, T. H., & Potti, S. (2020). *Browse PHP Answers by Framework*.

Code Grepper — Retrieved from https://www.codegrepper.com/code-examples/php

Grepper is a google chrome extension created to make programmers' lives easier when searching for code snippets.

Jacob: (2020, March 24). *PHP E_ERROR | Beamtic*.

PHP E_ERROR — Retrieved from "https://beamtic.com/e-error-php."

Beamtic website and/or services hosted by Beamtic are offered under the following conditions free of charge.

All content is copyrighted, and it is not allowed to copy content and publish it elsewhere on the internet.

You are allowed to quote articles in other online material (It is considered fair use).

Joomla!® is a registered trademark of Open Source Matters, Inc.: (2015, April 15). *Joomla! Platform Manual*.

Joomla! Platform Manual — Retrieved from. "http://joomla.github.io/joomla-platform/?coding-standards/chapters/php.md."

Contents are made available under the Joomla! Electronic Documentation License (JEDL) unless otherwise stated.

Lockhart, Josh., and *PhilSturgeon*, P.: (2022, May 8). *PHP: The Right Way*.

PHP: The Right Way — Retrieved from "https://phptherightway.com/#code_style_guide."

An easy-to-read, quick reference for PHP best practices, accepted coding standards, and links to authoritative PHP tutorials around the web.

PHP The Right Way is licensed under a "Creative Commons Attribution - NonCommercial-ShareAlike 3.0 Unported License".

Paul G. Allen School of Computer Science and Engineering.: (2017, April 10). *CSE 154 Unofficial Style Guide*.

CSE 154 Style Guide — Retrieved from: "https://courses.cs.washington.edu/courses/cse154/17sp/styleguide/index.html".

PHP Coding Standards.: (n.d.). *Backdrop CMS Documentation*.

PHP Coding Standards — Retrieved from: "https://docs.backdropcms.org/php-standards"

Backdrop is a free and Open-Source Content Management System that helps you build modern, comprehensive websites for businesses and non-profits.

The PHP Documentation Group.: © 1997-2022 (2022, May 9). *The PHP Manual*.

PHP Manual — Retrieved from: "https://www.php.net/manual/en."

PHP: (© 2001-2022) The PHP Group.: (2019, February 16). *pear Coding Standards Documentation*.

Coding Standards Documentation — Retrieved from: "https://pear.php.net/manual/en/standards.php."

Gautam.: (2019, July 26). *PHP7 Error Reporting (a developers' guide)*. © 2022 CodeClouds.

The complete guide to error reporting in PHP7 — Retrieved from: "https://www.codeclouds.com/blog/php7-error-reporting-a-2019-developers-guide".

CodeCloud offers CRM solutions, Creative Design, Web Development, App Development, CMS, E-Commerce, and I.T. services.

O'Phinney, M. W. O. and Kherlakian, M (2020, January 30). *Zend by perforce. PHP Development*.

Zend by perforce — Retrieved from "https://www.zend.com."

Sechrest, R.: (2013, July 17). *PHP style guide with coding standards and best practices. — GitHub*.

PHP style guide with coding standards and best practices. - php-style-guide.md. — Retrieved from: "https://gist.github.com/ryansechrest/8138375".

When code is put on GitHub, the author retains all the copyright. However, GitHub is granted a license to host the code. The author also allows GitHub users a set of rights, namely, looking at and forking the author's repositories.

Vanilla Foundation Theme: © 2022. (2021, October 29). *Coding Standard - PHP - Vanilla Success*.

Vanilla Success — Retrieved from: "https://success.vanillaforums.com/kb/articles/225-coding-standard-php".

W3 Schools | Refsnes Data. (1998, September 9). *PHP Reference*.

PHP Reference — Retrieved from "https://www.w3schools.com/php/php_ref_overview.asp".

W3Schools is a Freemium educational website for learning to code online.

W3Schools offers free online tutorials, references, and exercises in all the major languages of the web. Covering popular subjects like HTML, CSS, JavaScript, Python, SQL, Java, and many more.

WordPress Foundation. (2013, November 13). *PHP Coding Standards | Coding Standards Handbook | WordPress Developer*.

WordPress Developer Resources — Retrieved from "https://developer.wordpress.org/coding-standards/wordpress-coding-standards/php."

WordPress software is licensed under the General Public License (GPLv2 or later) from the Free Software Foundation.

Some parts of the WordPress code structure for PHP markup are inconsistent in their style, and WordPress is working to improve this gradually.

Made by awesome contributors. (2016, June 28). *Manual - Documentation — Zend Framework 2 2.4.9 documentation*.

Zend Framework Coding Standard for PHP — Retrieved from "https://framework.zend.com/manual/2.4/en/ref/coding.standard.html".

Zakharchenko, A. Z. (2017, August 5). GitHub - kettanaito/naming-cheatsheet: Naming-Cheatsheet

Naming-Cheatsheet — Retrieved from https://github.com/kettanaito/naming-cheatsheet

Comprehensive language-agnostic guidelines on variables naming. Home of the A/HC/LC pattern.

▲ References

23.3. WEBSITES

Research Websites	
Backdrop CMS Documentation - PHP Coding Standards	
Grepper	Joomla! Platform Manual
PHP Coding Guidelines & Best Practices — Flow Framework 7.3.x documentations	PHP Best Practices
PHP Coding Standards - Coding Standards Handbook - WordPress Developer Resources	PHP E_ERROR – Beamtic
PHP Manual	PHP Reference - W3 Schools
PHP style guide with coding standards and best practices	PHP- The Right Way
PSR-12- Extended Coding Style – PHP	Pear PHP Standards
Specbee - Introduction to PHP Standard Recommendation (PSR)	Standard PHP Library (SPL)
The complete guide to error reporting in PHP7	
Vanilla - Coding Standard - PHP	phpDocumentor

Table 8: Research Websites

▲ References

23.4. ACKNOWLEDGMENT

I thank the following individuals for their expertise and support throughout the writing of this manuscript:

Michael Bennet,	Service Delivery
Nelson Blocker,	Service Delivery
Raghib Qureshi,	Utility System/Engineering Specialist
Rhonda Ballew,	Service Delivery
Sean McGinnis,	Computer Audit Analyst
Todd Chambery,	Service Delivery

NOTE:

You are clearly a word lover, since this document contains 28,423 words!