Coding Standards

March 2025

Contents

Coding Standards

1. Purpose

This document provides a comprehensive set of coding standards for writing clean, readable, maintainable, and standardized PHP code. These conventions are derived from widely accepted standards such as PSR-12, WordPress Coding Standards, Drupal Coding Standards, PEAR Coding Standards and other industry best practices. Adhering to these guidelines ensures consistency across projects and improves collaboration among developers.

2. General Principles

- **Consistency:** Uniformity in style and naming reduces cognitive overhead.
- **Clarity:** Code should be self-explanatory with minimal need for external explanation.
- **Maintainability:** Structure and documentation should support long-term updates and debugging.
- **Compatibility:** Adhere to modern PHP versions (e.g., PHP 8.0+) while maintaining backward compatibility where necessary.

3. File Structure

3.1 File Naming

- Files should use lowercase letters with words separated by underscores (`snake_case`).
- For files containing classes, the file name must match the class name exactly, following PSR-4 autoloading conventions (e.g., `UserProfile` class in `user_profile.php`).
- Non-class files (e.g., scripts or templates) should describe their purpose (e.g., `config_loader.php`).

3.2 PHP Tags

- Use the full opening tag `<?php` at the start of every PHP file.
- Omit the closing tag `?>` in files containing only PHP code to prevent accidental whitespace or output, as recommended by PSR-12 and Drupal standards.

3.3 Namespace and Use Statements

- Declare a single namespace at the top of the file, immediately following the opening `<?php` tag or `declare` statement.

- Use use statements to import classes, functions, or constants, grouped by type (classes, functions, constants) and sorted alphabetically.
- Avoid leading backslashes in use statements (e.g., `use MyProject\ClassName;` not `use \MyProject\ClassName;`).
- Example:

```
<?php
namespace MyProject\Utilities;

use MyProject\Models\User;
use function MyProject\Helpers\logError;
use const MyProject\Constants\MAX_USERS;
```

3.4 Declare Statements

- Use `declare(strict_types=1);` to enforce strict type checking, placed immediately after the opening tag or file-level docblock, as per PSR-12.
- Example:

```
<?php
declare(strict_types=1);

function add(int $a, int $b): int {
    return $a + $b;
}
```

4. Coding Style

4.1 Indentation

- Use 4 spaces for indentation, as mandated by PSR-12. Do not use tabs to ensure consistency across editors and platforms.
- Example:

```
function processData($data) {
    if ($data) {
        return true;
    }
    return false;
}
```

4.2 Line Length

- Aim for a soft limit of 120 characters per line, with a recommendation to split lines exceeding 80 characters into multiple lines for readability (PSR-12, WordPress, Drupal).
- Split long lines logically (e.g., after operators or commas).
- Example:

```
$query = "SELECT * FROM users WHERE status = 'active' AND created_at > '2023-01-01' "
   . "ORDER BY created_at DESC";
```

4.3 Whitespace

- No trailing whitespace at the end of lines.
- Use one space after commas in argument lists and around binary operators (e.g., `$a + $b`).
- No spaces around unary operators (e.g., `++$i`).
- One blank line between logical blocks (e.g., methods, control structures).
- Example:

```
$sum = $a + $b;
$items = [1, 2, 3];
```

4.4 Braces

- Use braces for all control structures (e.g., `if`, `for`, `while`), even for single-line statements, as per PSR-12 and WordPress.
- Opening brace on the same line as the control structure, closing brace on its own line, aligned with the start of the control structure.
- Example:

```
if ($condition) {
   echo "True";
} else {
   echo "False";
}
```

4.5 Quotes

- Use single quotes (`'`) for static strings without variables or escape sequences.
- Use double quotes (`"`) for strings containing variables or escape sequences (e.g., `\n`).
- Example:

```
$static = 'Hello World';
$dynamic = "Hello, $name\n";
```

5. Naming Conventions

5.1 Variables

- Use camelCase (e.g., `$userName`) or snake_case (e.g., `$user_name`), with consistency enforced within the project.
- WordPress prefers snake_case; PSR-12 is neutral but emphasizes consistency.
- Example:

```
$firstName = 'John'; // camelCase
$first_name = 'John'; // snake_case
```

5.2 Functions

- Use camelCase (e.g., `getUserData`) or snake_case (e.g., `get_user_data`), starting with a lowercase letter.
- WordPress uses snake_case; PSR-12 allows either but requires consistency.
- Example:

```
function calculateTotal() {} // camelCase
function calculate_total() {} // snake_case
```

5.3 Classes

- Use PascalCase (e.g., `UserProfile`), as per PSR-12 and Drupal, to distinguish classes from other identifiers.
- Example:

```
class DatabaseConnection {}
```

5.4 Constants

- Use uppercase letters with underscores (e.g., `MAX_USERS`), as per PSR-12 and WordPress.
- Example:

```
const DEFAULT_TIMEOUT = 30;
define('API_KEY', 'xyz123');
```

5.5 File Names

- Use lowercase with underscores (e.g., `database_connection.php`), matching the class name for class files (PSR-4).

- Example: `user_profile.php` for `UserProfile`.

6. Syntax Rules

6.1 Control Structures

- Use `elseif` instead of `else if`, as per PSR-12 and WordPress.
- Always include braces, even for single-line statements.
- In `switch` statements, include a `break` unless fall-through is intentional (documented with a comment).
- Example:

```
if ($value > 0) {
   return 'Positive';
} elseif ($value < 0) {
   return 'Negative';
} else {
   return 'Zero';
}

switch ($type) {
   case 'admin':
      // Fall-through intentional
   case 'editor':
      return true;
   default:
      return false;
}
```

6.2 Operators

- Use strict comparison operators (`===`, `!==`) instead of loose (`==`, `!=`) to avoid type juggling, as recommended by WordPress and PSR-12.
- Use the ternary operator (`?:`) sparingly; prefer `if-else` for complex conditions.
- Example:

```
if ($value === null) {
   $result = 'Empty';
} else {
   $result = 'Set';
}
```

6.3 Arrays

- Use long array syntax (`array()`) for compatibility with older PHP versions, though short syntax (`[]`) is acceptable for PHP 5.4+.
- For multi-line arrays, place each element on its own line with a trailing comma, as per Drupal and WordPress.
- Example:

```
$data = array(
   'name' => 'John',
   'age' => 30,
);
```

6.4 Strings

- Use single quotes for static strings, double quotes for interpolated strings.
- Concatenate with `.` operator, with spaces around it for readability.
- Example:

```
$message = 'Hello' . ' ' . $name;
```

6.5 Functions and Methods

- No space between the function/method name and the opening parenthesis.
- One space after commas in argument lists.
- Example:

```
function process($input, $options = []) {
   return $input;
}
```

7. Documentation

7.1 Docblocks

- Use PHPDoc format for all documentation, including `@param`, `@return`, `@throws`, and other relevant tags.
- Place docblocks immediately above the element they describe (e.g., class, function).
- For classes, include `@package` and optionally `@author` or `@since`.
- Example:

```php
/**
 * Retrieves user data by ID.
 *
 * @param int $userId The ID of the user.
 * @param bool $includeMeta Whether to include metadata.
 * @return array User data.
 * @throws InvalidArgumentException If userId is invalid.
 */
function getUserData(int $userId, bool $includeMeta = false): array {
    // Implementation
}
```

7.2 Inline Comments

- Use `//` for single-line comments, placed above or at the end of the line.
- Use `/* */` for multi-line comments, aligned with the code block.
- Example:

```php
// Check if user exists
if ($user) {
    /* Multi-line comment explaining
       complex logic below */
    $result = process($user);
}
```

References

- PSR-12 Extended Coding Style Guide
- WordPress PHP Coding Standards
- Drupal PHP Coding Standards
- PEAR Coding Standards