PHP Coding Standards

For Developers

Presented On 13-Feb-2014
By,
Sangeeta Arora – Technical Lead | CIPL
sangeeta.arora@classicinformatics.com

Agenda

- **Introduce PSR**
  - What is PSR?
  - Different types of PSR
- **Introduce DocBlocks**
  - What is DocBlock?
  - Different DocBlock Tags
- **Introduce PHP_CodeSniffer**
  - What is PHP_CodeSniffer?

PSR?

- **PSR stands for *PHP Standards Recommendation*.**
- **Coding standards are guidelines for code style and documentation.**
- **Purpose of coding standards is to make it so that source code looks as if it's written by a single developer as it sets a level of expectations for contributing developers.**
- **Five Types of PSR**
  - **PSR – 0 – Autoloading Standard**
  - **PSR – 1 – Basic Coding Standard**
  - **PSR – 2 – Coding Style Guide**
  - **PSR – 3 – Logger Interface**
  - **PSR – 4 – Improved Autoloading**
- **We will discuss PSR – 0, PSR – 1 and PSR – 2 in the further slides.**
- **We will discuss PSR – 3 and PSR – 4 in future presentations.**

PSR – 0 – Autoloading Standard

A standard recommendation that describes "the mandatory requirements that must be adhered to for autoloader interoperability.

PSR – 0 – Autoloading Standard

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

PSR – 0 – Examples

- \Doctrine\Common\IsolatedClassLoader => /path/to/project/lib/vendor/Doctrine/Common/IsolatedClassLoader.php

- \Symfony\Core\Request => /path/to/project/lib/vendor/Symfony/Core/Request.php.

- \Zend\Acl => /path/to/project/lib/vendor/Zend/Acl.php.

- \Zend\Mail\Message => /path/to/project/lib/vendor/Zend/Mail/Message.php.

PSR – 1 – Basic Coding Standard

It focuses on a Basic Coding Standard.

PSR – 1 – Basic Coding Standard

- Files MUST use only <?php and <?= tags.
- Files MUST use only UTF-8 (without BOM) character encoding for PHP code.
- Files SHOULD *either* declare symbols (classes, functions, constants, etc.) *or* cause side-effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow PSR-0.
- **Class** names MUST be declared in **StudlyCaps**.
- Class **constants** MUST be declared in all **upper case** with underscore separators, if required. E.g. const VERSION; const DATE_APPROVED;
- Class **method** names MUST be declared in **camelCase**.

PSR – 2 – Coding Style Guide

Its purpose is to have a single style guide for PHP code that results in uniformly formatted shared code.

PSR – 2 – Coding Style Guide

- **Basic Coding Standard**
  - Code MUST follow PSR-1.

- **Files**
  - All PHP files MUST use the Unix LF (linefeed) line ending.
  - All PHP files MUST end with a single blank line.
  - The closing ?> tag MUST be omitted from files containing only PHP.

- **Lines**
  - There MUST NOT be a hard limit on line length.
  - The soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.
  - There MUST NOT be trailing whitespace at the end of non-blank lines.
  - Blank lines MAY be added to improve readability and to indicate related blocks of code.
  - There MUST NOT be more than one statement per line.

PSR – 2 – Coding Style Guide …contd.

- **Indenting**
  - Code MUST use 4 spaces for indenting, and MUST NOT use tabs for indenting.
- **Keywords and True/False/Null**
  - PHP **keywords** MUST be in lower case.
  - The PHP **constants** true, false, and null MUST be in lower case.
- **Namespace and Use Declarations**
  - There MUST be one blank line after the **namespace** declaration.
  - All **use** declarations MUST go after the **namespace** declaration.
  - There MUST be one use keyword per declaration..
  - There MUST be one blank line after the **use** block.

```php
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

// ... additional PHP code ...
```

PSR – 2 – Coding Style Guide …contd.

- **Extends and Implements**
  - The **extends** and **implements** keywords MUST be declared on the same line as the class name. e.g.

```
class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constants, properties, methods
}
```

  - Lists of implements MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

```
class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constants, properties, methods
}
```

  - Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.
- **Properties**
  - Visibility MUST be declared on all properties.

PSR – 2 – Coding Style Guide …contd.

- **Properties**
  - There MUST NOT be more than one property declared per statement.
  - The **var** keyword MUST NOT be used to declare a property.
  - Property names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility. e.g.

```php
<?php
namespace Vendor\Package;

class ClassName
{
    public $foo = null;
}
```

- **Methods**
  - Visibility MUST be declared on all methods.
  - Method names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.
  - Method names MUST NOT be declared with a space after the method name.
  - The opening brace MUST go on its own line, and the closing brace MUST go on the next line following the body.

PSR – 2 – Coding Style Guide …contd.

- **Methods**
  - There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. e.g.

```php
<?php
namespace Vendor\Package;

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

- **Method Arguments**
  - There MUST NOT be a space before each comma.
  - There MUST be one space after each comma.
  - Method arguments with default values MUST go at the end of the argument list.

PSR – 2 – Coding Style Guide …contd.

- **Method Arguments**

```php
<?php
namespace Vendor\Package;

class ClassName
{
    public function foo($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

Example of Arguments in the same line

```php
<?php
namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // method body
    }
}
```

Example of Arguments not in the same line, when have a long list of arguments.

PSR – 2 – Coding Style Guide …contd.

- **Qualifiers (abstract, final, and static)**
  - When present, the **abstract** and **final** declarations MUST precede the visibility declaration.
  - When present, the **static** declaration MUST come after the visibility declaration.

```php
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

PSR – 2 – Coding Style Guide …contd.

- **Method and Function Calls**
  - There MUST NOT be a space between the method or function name and the opening parenthesis.
  - There MUST NOT be a space after the opening parenthesis.
  - There MUST NOT be a space before the closing parenthesis.
  - In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

```php
<?php
bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

Example of Arguments in the same line

```php
<?php
$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

Example of Arguments not in the same line, when have a long list of arguments.

PSR – 2 – Coding Style Guide …contd.

- **Control Structures**
  - There MUST be one space after the control structure keyword.
  - There MUST NOT be a space after the opening parenthesis.
  - There MUST NOT be a space before the closing parenthesis.
  - There MUST be one space between the closing parenthesis and the opening brace.
  - The structure body MUST be indented once.
  - The closing brace MUST be on the next line after the body.

PSR – 2 – Coding Style Guide …contd.

- **if-elseif-else**
  - The keyword **elseif** SHOULD be used instead of **else if**.
  - **else** and **elseif** are on the same line as the closing brace from the earlier body.
  - Note the placement of parentheses, spaces, and braces.

```php
<?php
if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

PSR – 2 – Coding Style Guide …contd.

- **switch and case**
  - The **case** statement MUST be indented once from **switch**.
  - The break keyword (or other terminating keyword) MUST be indented at the same level as the case body.
  - There MUST be a comment such as // no break when fall-through is intentional in a non-empty case body.
  - Note the placement of parentheses, spaces, and braces.

```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;
}
```

PSR – 2 – Coding Style Guide …contd.

- **while and do-while**
  - Note the placement of parentheses, spaces, and braces for **while** statement.

```php
<?php
while ($expr) {
    // structure body
}
```

  - Note the placement of parentheses, spaces, and braces for **do-while** statement.

```php
<?php
do {
    // structure body;
} while ($expr);
```

PSR – 2 – Coding Style Guide …contd.

- **for and foreach**
  - Note the placement of parentheses, spaces, and braces for **for** statement.

```php
<?php
for ($i = 0; $i < 10; $i++) {
    // for body
}
```

  - Note the placement of parentheses, spaces, and braces for **foreach** statement.

```php
<?php
foreach ($iterable as $key => $value) {
    // foreach body
}
```

PSR – 2 – Coding Style Guide …contd.

- **try-catch**
  - Note the placement of parentheses, spaces, and braces for **try-catch** statement.

```php
<?php
try {
    // try body
} catch (FirstExceptionType $e) {
    // catch body
} catch (OtherExceptionType $e) {
    // catch body
}
```

PSR – 2 – Coding Style Guide …contd.

- **In-Code Documentation**
  - All PHP files should include a DocBlock header.
  - All classes should include a DocBlock for the class definition.
  - All class constants, properties and methods should include a DocBlock to explain their purpose.
  - All functions should include a DocBlock to explain their purpose.

PSR – 3 – Logger Interface

PSR-3 describes a shared logging interface which improves reusability.

PSR – 4 – Improved Autoloading

The main goal of PSR-4 is to remove the remnants of PSR-0 and the pre-5.3 days completely and allow for a more concise folder structure.

DocBlocks

DocBlock is a multi-line C-style comment used to document a block of code.

DocBlocks

- It is a multi-line C-style comment used to document a block of code.
- "DocComment" is a special type of comment which MUST
  - It begins with the character sequence /** followed by a whitespace character
  - It has an asterisk at the start of each line.
  - end with */
  - have zero or more lines in between
- DocBlocks consist of three sections:-
  - short description
  - long description
  - tags
- Short description is a concise description terminated by either a new-line or a period.
- Long description is where the bulk of the documentation goes; it can be multi-line and as long you wish.

DocBlocks …contd.

- **Tags**
  - The tags are used to specify additional information, such as the expected parameters and their type.
  - Each tag starts on a new line, followed by an at-sign (@) and a tag-name followed by white-space and meta-data (including a description).

```
@param string $argument1 This is a parameter.
```

The above tag consists of a name ('param') and meta-data ('string $argument1 This is a parameter.') where the meta-data is split into a "Type" ('string'), variable name ('$argument') and description ('This is a parameter.').

DocBlocks …contd.

- **Different Types of Tags**
  - @api
  - @author
  - @copyright
  - @deprecated
  - @example
  - @global
  - @internal
  - @license
  - @method
  - @package
  - @param
  - @property
  - @return
  - @see
  - @since
  - @struct
  - @throws
  - @todo
  - @type
  - @uses
  - @version

DocBlocks …contd.

- **@api** – It is used to declare "Structural Elements" as being suitable for consumption by third parties.

```
/**
 * This method will not change until a major release.
 *
 * @api
 *
 * @return void
 */
```

- **@author** – The @author tag is used to document the author of any "Structural Element".

```
/**
 * @author My Name
 * @author My Name <my.name@example.com>
 */
```

- **@copyright** – The @copyright tag is used to document the copyright information of any "Structural Element".

```
/**
 * @copyright 1997-2005 The PHP Group
 */
```

DocBlocks …contd.

- **@deprecated** – The @deprecated tag is used to indicate which 'Structural elements' are deprecated and are to be removed in a future version.

```
/**
 * @deprecated
 * @deprecated 1.0.0
 * @deprecated No longer used by internal code and not recommended.
 * @deprecated 1.0.0 No longer used by internal code and not recommended.
 */
```

- **@example** – The @example tag is used to link to an external source code file which contains an example of use for the current "Structural element".

```
/**
 * Counts the number of items.
 * {@example http://example.com/foo-inline.https:2..8}
 *
 * @example http://example.com/foo.phps
 *
 * @return integer Indicates the number of items.
 */
function count()
{
    <...>
}
```

DocBlocks …contd.

- **@global** – The @global tag is used to denote a global variable or its usage.

```
@global ["Type"] [name]
@global ["Type"] [description]
```

- **@license** – The @license tag is used to indicate which license is applicable for the associated 'Structural Elements'.

```
/**
 * @license MIT
 * @license http://www.spdx.org/licenses/MIT MIT License
 */
```

- **@method** – The @method allows a class to know which 'magic' methods are callable.

```
@method [return type] [name]([type] [parameter], [...]) [description]
```

```
/**
 * @method string getString()
 * @method void setInteger(integer $integer)
 * @method setString(integer $integer)
 */
class Child extends Parent
{
```

DocBlocks …contd.

- **@package** – The @package tag is used to categorize "Structural Elements" into logical subdivisions.

```
/**
 * @package PSR\Documentation\API
 */
```

- **@param** – The @param tag is used to document a single parameter of a function or method.

```
@param ["Type"] [name] [<description>]
```

```
/**
 * Counts the number of items in the provided array.
 *
 * @param mixed[] $array Array structure to count the elements of.
 *
 * @return int Returns the number of elements.
 */
```

- **@property** – The @property tag is used in the situation where a class contains the __get() and __set() magic methods and allows for specific names.

```
@property ["Type"] [name] [<description>]
```

DocBlocks …contd.

- **@return** – The @return tag is used to document the return value of functions or methods.

```
@return <"Type"> [description]
```

```
/**
 * @return integer Indicates the number of items.
 */
function count()
{
    <...>
}
```

- **@see** – The @see tag can be used to define a reference to other "Structural Elements" or to a URI.

```
/**
 * @see number_of() :alias:
 * @see MyClass::$items           For the property whose items are counted.
 * @see MyClass::setItems()       To set the items for this collection.
 * @see http://example.com/my/bar Documentation of Foo.
 *
 * @return integer Indicates the number of items.
 */
function count()
{
    <...>
}
```

DocBlocks …contd.

- **@since** – The @since tag is used to denote when an element was introduced or modified, using some description of "versioning" to that element.
- The @since tag SHOULD NOT be used to show the current version of an element, the @version tag MAY be used for that purpose.

```
/**
 * This is Foo
 * @version MyApp 2.1.7
 * @since 2.0.0 introduced
 */
class Foo
{
```

- **@todo** – The @todo tag is used to indicate whether any development activities should still be executed on associated "Structural Elements".

```
/**
 * Counts the number of items in the provided array.
 *
 * @todo add an array parameter to count
 *
 * @return int Returns the number of elements.
 */
function count()
```

DocBlocks …contd.

- **@throws –** The @throws tag is used to indicate whether "Structural Elements" throw a specific type of exception.

```
/**
 * Counts the number of items in the provided array.
 *
 * @param mixed[] $array Array structure to count the elements of.
 *
 * @throws InvalidArgumentException if the provided argument is not of type
 *     'array'.
 *
 * @return int Returns the number of elements.
 */
function count($items)
{
    <...>
}
```

- **@version –** The @version tag is used to denote some description of "versioning" to an element.

```
@version ["Semantic Version"] [<description>]
```

DocBlocks …contd.

- **@type –** The @type tag defines which type of data is represented by a value of a Constant, Property or Variable.

```php
/** @type int $int This is a counter. */
$int = 0;

// there should be no docblock here
$int++;
```

- **@uses –** The @uses tag describes whether any part of the associated "Structural Element" uses, or consumes, another "Structural Element" or a file that is situated in the current project.

```php
/**
 * @uses \SimpleXMLElement::__construct()
 */
function initializeXml()
```

```php
/**
 * @uses MyView.php
 */
function executeMyView()
```

PHP_CodeSniffer

What is PHP_CodeSniffer?

- PHP_CodeSniffer (*PHPCS*) is a PHP5 script that tokenizes PHP, JavaScript and CSS files to detect violations of a defined coding standard.
- It is an essential development tool that ensures your code remains clean and consistent.
- It can also help prevent some common semantic errors made by developers.
- It allows you to not only ensure that your code follows the PSR standards, but it also uses PHP's linter to check for syntax errors.
- Multiple coding standards can be used within PHP_CodeSniffer so that the one installation can be used across multiple projects.
- The default coding standard used by PHP_CodeSniffer is the PEAR coding standard.

Q & A

Thank You