CHAPTER 3

Coding Standards

Coding standards are set definitions of how to structure your code in any given project. Coding standards apply to everything from naming conventions and spaces to variable names, opening and closing bracket placement, and so on. We use coding standards as a way to keep code uniform throughout an entire project, no matter the number of developers that may work on it. If you've ever had to work on a project that has no consistency in variable names, class or method names, and so on, then you've experienced what it is like to work through code that didn't adhere to coding standards. Imagine now how much easier the code would be to both follow and write if you knew the exact way that it should be structured throughout the entire project?

There are a number of PHP code standards that have waxed and waned in popularity and prevalence throughout the years. There are the PEAR Standards, which are very detailed; the Zend Framework Standard, which is promoted by Zend; and within the last five years or so, we've seen standards created by a board of developers called PHP-FIG.

Although this chapter is focused on PHP-FIG, there are no right or wrong answers on the standards you pick. The important takeaway from this chapter is that it's important to at least follow some standards! Even if you create your own, or decide to create your own variation that deviates a bit from an already popular existing one, just pick one and go with it.

We'll take a look at the PHP-FIG standards body and the standards they've developed and promote. We'll also look at the tools you can use to help enforce the proper use of given standards throughout your project's development.

A Look at PHP-FIG

PHP-FIG (`php-fig.org`) is the PHP Framework Interoperability Group, which is a small group of people that was originally formed at the phptek conference in 2009 to create a standards body for PHP frameworks. It has grown from five founding members to twenty, and has published several standards.

The PHP Standards Recommendations are:

- PSR-0 - Autoloader Standard
- PSR-1 - Basic Coding Standard
- PSR-2 - Coding Style Guide

C. Russell, *PHP Development Tool Essentials*, DOI 10.1007/978-1-4842-0683-6_3

- PSR-3 - Logger Interface
- PSR-4 - Autoloader Standard

For this chapter we'll look at PSR-1 and PSR-2, which stand for PHP Standard Recommendations 1 and 2. These standards are fairly straightforward, easy to follow, and could even be used as a solid foundation to create your own coding standards, if you wanted.

PSR-1 – Basic Coding Standard

The full spec of this standard can be found at `http://www.php-fig.org/psr/psr-1/`. This is the current standard as of the time of writing. This section is meant to give you a general overview of the standard and some basic examples of following it. The standard is broken down into the following structure:

- Files
- Namespace and Class Names
- Class Constants, Properties, and Methods

Files

The standards definitions for Files under PSR-1 are described in this section.

PHP Tags

PHP code must use `<?php` tags or the short echo tag in `<?=` format. No other tag is acceptable, even if you have short tags enabled in your PHP configuration.

Character Encoding

PHP code must use only UTF-8 without the byte-order mark (BOM). For the most part, this isn't one you have to worry about. Unless you're writing code in something other than a text editor meant for coding (such as SublimeText, TextWrangler, Notepad++, etc.) or an IDE, it's not something that should just automatically be included. The biggest reason this isn't allowed is because of the issues it can cause when including files with PHP that may have a BOM, or if you're trying to set headers, because it will be considered output before the headers are set.

Side Effects

This standard says that a PHP file should either declare new symbols (classes, functions, constants, etc.) or execute logic with side effects, but never both. They use the term *side effects* to denote logic executed that's not directly related to declaring a class, functions or methods, constants, etc. So, in other words, a file shouldn't both declare a function AND execute that function, as seen in the example that follows, thus enforcing better code separation.

```
<?php

// Execution of code
myFunction();

// Declaration of function
function myFunction() {
  // Do something here
}
```

Namespace and Class Names

The standards definitions for namespaces and class names under PSR-1 are as follows:

- Namespaces and classes must follow an autoloading PSR, which is currently either PSR-0, the Autoloading Standard, or PSR-4, the Improved Autoloading Standard. By following these standards, a class is always in a file by itself (there are not multiple classes declared in a single file), and it includes at least a namespace of one level.
- Class names must be declared using `StudlyCaps`.

Here is an example of how a class file should look:

```
<?php

namespace Apress\PhpDevTools;

class MyClass
{
  // methods here
}
```

Class Constants, Properties, and Methods

Under this standard, the term *class* refers to all classes, interfaces, and traits.

Constants

Class constants must be declared in all uppercase using underscores as separators.

Properties

The standards are fairly flexible when it comes to properties within your code. It's up to you to use `$StudlyCaps`, `$camelCase`, or `$under_score` property names; you can mix them if they are outside of the scope of each other. So, if your scope is vendor, package, class, or method level, just be consistent within that given scope. However, I would argue that it's

really best to find one and stick to it throughout all of your code. It will make the uniformity and readability of your code much easier as you switch through classes and such.

Methods

Method names must be declared using `camelCase()`. Let's take a look at this standard in use:

```
<?php

namespace Apress\PhpDevTools;

class MyClass
{
    const VERSION = '1.0';

    public $myProperty;

    public function myMethod()
    {
        $this->myProperty = true;
    }
}
```

This wraps up all there is to the PSR-1 Basic Coding Standard. As you can see, it's really straightforward, easy to follow, and easy to get a handle on just after your first read-through of it. Now, let's look at PSR-2, which is the Coding Style Guide.

PSR-2 – Coding Style Guide

This guide extends and expands on PSR-1 and covers standards for additional coding structures. It is a longer read than PSR-1. This guide was made by examining commonalities among the various PHP-FIG member projects. The full spec of this standard can be found at http://www.php-fig.org/psr/psr-2/. As with PSR-1, this is the current standard as of the time of writing. This section is meant to give you a general overview of the standard and some basic examples of following it. The standard is broken down into the following structure:

- General
- Namespace and Use Declarations
- Classes, Properties, and Methods
- Control Structures
- Closures

General

In addition to the following rules, in order to be compliant with PSR-2 the code must follow all of the rules outlined in PSR-1 as well.

Files

All PHP files must use the Unix linefeed line ending, must end with a single blank line, and must omit the close `?>` tag if the file only contains PHP.

Lines

The standards for lines follow a number of rules. The first three rules deal with line lengths:

- There must not be a hard limit on the length of a line.
- There must be a soft limit of 120 characters.
- Lines should not be longer than 80 characters, and should be split into multiple lines if they go over 80 characters.

The last two rules seem to contradict themselves a bit, but I believe the reasoning behind them is that, generally, 80 characters has been the primary standard for lines of code. There are many, many arguments around why it's 80, but most agree it's the most readable length across devices and screen sizes. The soft limit of 120 is more of a visual reminder that you've passed 80 to 120, which is also easily readable on most IDEs and text editors viewed on various screen sizes, but deviates past the widely accepted 80. The no hard limit rule is there because there may be occasional scenarios where you need to fit what you require on a line and it surpasses both the 80- and 120-character limits.

The remaining line rules are:

- 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. This one can really help, so that all of your code does not run together.
- You can only have one statement per line.

Indentation

This rule states that you must use four spaces and never use tabs. I've always been a proponent of using spaces over tabs, and most any code editor or IDE can easily map spaces to your tab key, which just makes this rule even easier to follow.

Keywords and true, false, and null

This rule states that all keywords must be in lowercase as must the constants `true`, `false`, and `null`.

Namespace and Use Declarations

This standard states the following in regards to using namespaces and declarations of namespaces:

- There must be one blank line after the namespace is declared.
- Any use declarations must go after the namespace declaration.
- You must only have one use keyword per declaration. So even though you can easily define multiple declarations in PHP separated by a comma, you have to have one per line, with a use declaration for each one.
- You must have one blank line after the use block.

Classes, Properties, and Methods

In these rules, the term *class* refers to all classes, interfaces, and traits.

Classes

- The `extends` and `implements` keywords must be declared on the same line as the class name.
- The opening brace for the class must go on its own line, and the closing brace must appear on the next line after the body of your class.
- Lists of implements may be split across multiple lines, where each subsequent line is indented once. When doing this, the first item in the list must appear on the next line, and there must only be one interface per line.

Properties

- Visibility (public, private, or protected) must be declared on all properties in your classes.
- The `var` keyword must not be used to declare a property.
- There must not be more than one property declared per statement.
- Property names should not be prefixed with a single underscore to indicate protected or private visibility. This practice is enforced with the Pear coding standards, so there is a good chance you've seen code using this method.

Methods

- Visibility (public, private, or protected) must be declared on all methods in your classes, just like with properties.
- Method names should not be prefixed with a single underscore to indicate protected or private visibility. Just as with properties, there's a good chance you've seen code written this way; however, it is not PSR-2 compliant.
- Method names must not be declared with a space after the method name, and the opening brace must go on its own line and the closing brace on the next line following the body of the method. No space should be used after the opening or before the closing parenthesis.

Method Arguments

- In your method argument list, there must not be a space before each comma, but there must be one space *after* each comma.
- Method arguments with default values must go at the end of the argument list.
- You can split your method argument lists across multiple lines, where each subsequent line is indented once. When using this approach, the first item in the list must be on the next line, and there must be only one argument per line.
- If you use the split argument list, the closing parenthesis and the opening brace must be placed together on their own line with one space between them.

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.

Method and Function Calls

When you make a method or function call, 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 or before the closing parenthesis. In the argument list, there must not be a space before each comma, but there must be one space *after* each comma.

The argument list may also be split across multiple lines, where each subsequent line is indented once. When doing this, the first item in the list must be on the next line, and there must be only one argument per line.

Control Structures

There are several general style rules for control structures. They are as follows:

- There must be one space after the control structure keyword.
- There must not be a space after the opening parenthesis or before the closing parenthesis.
- There must be one space between the closing parenthesis and the opening brace, and the closing brace must be on the next line after the body.
- The structure body must be indented once.
- The body of each structure must be enclosed by braces. This is definitely a very helpful rule, because it creates uniformity and increases readability with control structures and not having braces, even though it's allowed for single-line statements or when using the PHP alternate syntax, can sometimes lead to less readable code.

The next few rules are all basically identical for the following control structures. Let's look at simple examples of each of them.

if, elseif, else

This builds on the previous rule and states that a control structure should place `else` and `elseif` on the same line as the closing brace from the previous body. Also, you should always use `elseif` instead of `else if` so the keywords are all single words. For example, this is a fully compliant `if` structure:

```
<?php

if ($a === true) {

} elseif ($b === true {

} else {

}
```

switch, case

The `case` statement must be indented once from the `switch` keyword, and the `break` keyword or other terminating keyword (`return, exit, die`, etc.) 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. The following example is a compliant switch structure:

```
<?php

switch ($a) {
    case 1:
        echo "Here we are.";
        break;
```

```
    case 2:
        echo "This is a fall through";
        // no break
    case 3:
        echo "Using a different terminating keyword";
        return 1;
    default:
        // our default case
        break;
}
```

while, do while

The `while` and `do while` structures place the braces and spaces similarly to those in the `if` and `switch` structures:

```
<?php

while ($a < 10) {
    // do something
}

do {
    // something
} while ($a < 10);
```

for

The PSR-2 documentation shows that a `for` statement should be formatted as in the following example. One thing that is unclear based on what they have listed as well as based on their general rules for control structures is whether spaces are required between the `$i = 0` and the `$i < 10` in the example that follows. Removing the spaces and running it against PHP Code Sniffer with PSR-2 validation will result in it passing validation, so this is left up to you according to your preference. Both of the following examples are PSR-2 compliant:

```
<?php

for ($i = 0; $i < 10; $i++) {
    // do something
}

for ($j=0; $j<10; $i++) {
    // do something
}
```

foreach

A PSR-2 compliant `foreach` statement should be structured as in the following example. Unlike in the `for` statement, the space is required if you are separating the key and value pairs using the `=>` assignment:

```
<?php

foreach ($array as $a) {
    // do something
}

foreach ($array as $key => $value) {
    // do something
}
```

try, catch (and finally)

Last in the control structure rules is the `try catch` block. A `try catch` block should look like the following example. The PSR-2 standard doesn't include anything about the `finally` block (PHP 5.5 and later), but if using it, you should follow the same structure as in the `try` block:

```
<?php

try {
    // try something
} catch (ExceptionType $e) {
    // catch exception
} finally {
    // added a finally block
}
```

Closures

Closures, also known as anonymous functions, have a number of rules to follow for the PSR-2 standard. They are very similar to the rules that we have for functions, methods, and control structures. This is mostly due to closures being anonymous functions, so the similarities between them and functions and methods make them close to identical. The PSR-2 rules are as follows:

- Closures must be declared with a space after the `function` keyword and a space both before and after the `use` keyword.
- The opening brace must go on the same line, and the closing brace must go on the next line, following the body, just as with functions, methods, and control structures.

- There must not be a space after the opening parenthesis of the argument list or variable list, and there must not be a space before the closing parenthesis of the argument list or variable list. Again, this is the same as with functions and methods.
- There must not be a space before each comma in the argument list or variable list, and there must be one space after each comma in the argument list or variable list.
- Closure arguments with default values must go at the end of the argument list, just as with regular functions and methods.

Here are a few examples of closures that are PSR-2 compliant:

```
<?php

// Basic closure
$example = function () {
    // function code body
};

// Closure with arguments
$example2 = function ($arg1, $arg2) {
    // function code body
};

// Closure inheriting variables
$example3 = function () use ($var1, $var2) {
    // function code body
};
```

Just as with functions and methods, argument lists and variable lists may be split across multiple lines. The same rules that apply to functions and methods apply to closures.

Lastly, if a closure is being used directly in a function or method call as an argument, it must still follow and use the same formatting rules. For example:

```
<?php

$myClass->method(
    $arg1,
    function () {
        // function code body
    }
);
```

These rules conclude the PSR-2 Coding Style Guide. As you can see, they build on the basic rules set forth in PSR-1, and most of them build on rules from each other and share a number of commonalities.

Omissions from PSR-2

There are many elements that were intentionally omitted by the PSR-2 standard (although these items may eventually become part of the specification over time). These omissions include but are not limited to the following, according to the PSR-2 specification:

Declaration of global variables and global constants

Declaration of functions

Operations and assignment

Inter-line alignment

Comments and documentation blocks

Class name prefixes and suffixes

Best practices

Checking Coding Standards with PHP Code Sniffer

Coding standards are a great thing to have, and online resources, such as the documentation provided by PHP-FIG on PSR-1 and PSR-2, help aid you in making the correct choices so that your code is compliant. However, it's still easy to forget a rule or mistype something to make it invalid, or maybe you're part of a team and it's impossible to do code reviews on everyone's code to make sure all of their commits are compliant. This is where it's good to have a code validator that everyone can easily run, or that could even be incorporated into an automated process, to ensure all code is compliant with PSR-1 and PSR-2, or even with another coding standard that you choose.

A tool such as this exists, and it's called the PHP Code Sniffer, also referred to as PHP_CodeSniffer by Squizlabs. PHP_CodeSniffer is a set of two PHP scripts. The first is `phpcs`, which, when run, will tokenize PHP files (as well as JavaScript and CSS) to detect violations of a defined coding standard. The second script is `phpcbf`, which can be used to automatically correct coding standard violations.

PHP_CodeSniffer can be installed a number of different ways. You can download the Phar files for each of the two commands, you can install using Pear, or you can install using Composer. Here are the steps for each of these installation methods.

1. Downloading and executing Phar files:

```
$ curl -OL https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar

$ curl -OL https://squizlabs.github.io/PHP_CodeSniffer/phpcbf.phar
```

2. If you have Pear installed you can install it using the PEAR installer. This is done by the following command:

```
$ pear install PHP_CodeSniffer
```

3. Lastly, if you use and are familiar with Composer (using Composer is covered in Chapter 4) then you can install it system-wide with the following command:

```
$ composer global require "squizlabs/php_codesniffer=*"
```

■ **Note** The full online documentation for PHP CodeSniffer can be found at https://github.com/squizlabs/PHP_CodeSniffer/wiki.

Using PHP_CodeSniffer

Once you have installed PHP_CodeSniffer, you can use it either via command line or directly in some IDEs, such as PHP Storm or NetBeans. Using it from the command line is the quickest way to get started using it. You can use it to validate a single file or an entire directory.

■ **Note** One PHP_CodeSniffer prerequisite is that the PEAR package manager is installed on the machine.

Right now, you can find two files, named `invalid.php` and `valid.php`, in the "Chapter3" branch of the accompanying code repository for this book. We're going to test PHP_CodeSniffer against these files:

```
$ phpcs --standard=PSR1,PSR2 invalid.php

FILE: /Apress/source/invalid.php
----------------------------------------------------------------------
FOUND 10 ERRORS AFFECTING 5 LINES
----------------------------------------------------------------------
  3 | ERROR | [ ] Each class must be in a namespace of at least one
    |       |     level (a top-level vendor name)
  3 | ERROR | [x] Opening brace of a class must be on the line after
    |       |     the definition
  4 | ERROR | [ ] Class constants must be uppercase; expected VERSION
    |       |     but found version
  6 | ERROR | [ ] The var keyword must not be used to declare a
    |       |     property
  6 | ERROR | [ ] Visibility must be declared on property "$Property"
  8 | ERROR | [ ] Method name "ExampleClass::ExampleMethod" is not in
    |       |     camel caps format
  8 | ERROR | [ ] Expected "function abc(...)"; found "function abc
    |       |     (...)"
```

```
  8 | ERROR | [x] Expected 0 spaces before opening parenthesis; 1
    |       |     found
  8 | ERROR | [x] Opening brace should be on a new line
 11 | ERROR | [x] Expected 1 newline at end of file; 0 found
----------------------------------------------------------------------
PHPCBF CAN FIX THE 4 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------
```

From this output we see there are ten different errors that were detected when validating against the PSR-1 and PSR-2 standards. You can pass in different standards to be used for validating, and even pass multiple standards separated by a comma, as in this example using PSR1 and PSR2. Also, out of the ten errors, four were marked as able to be fixed automatically using the PHP Code Beautifier and Fixer, otherwise known as the `phpcbf` tool. We can now run `phpcbf` against the file and try the validation again to see if it fixes it:

```
$ phpcbf --standard=PSR1,PSR2 invalid.php
Changing into directory /Apress/source
Processing invalid.php [PHP => 52 tokens in 11 lines]... DONE in 4ms (4
fixable violations)
        => Fixing file: 0/4 violations remaining [made 3 passes]... DONE in 7ms
Patched 1 file
```

As you can see, `phpcbf` is used just like `phpcs` in that you can pass in a list of standards to use to correct for, and then the file name. Now, to run the validator on the file again:

```
$ phpcs --standard=PSR1,PSR2 invalid.php
FILE: /Apress/source/invalid.php
----------------------------------------------------------------------
FOUND 5 ERRORS AFFECTING 4 LINES
----------------------------------------------------------------------
 3 | ERROR | Each class must be in a namespace of at least one level
   |       | (a top-level vendor name)
 5 | ERROR | Class constants must be uppercase; expected VERSION but
   |       | found version
 7 | ERROR | The var keyword must not be used to declare a property
 7 | ERROR | Visibility must be declared on property "$Property"
 9 | ERROR | Method name "ExampleClass::ExampleMethod" is not in
   |       | camel caps format
----------------------------------------------------------------------
```

Running the test after running `phpcbf`, we see it actually fixed one additional issue when it fixed another, so now there are only five errors found. Now, if we run it against our `valid.php` file, which is completely valid, we'll see what a valid result looks like:

```
$ phpcs --standard=PSR1,PSR2 valid.php
$
```

With a 100 percent valid file, there is no output from `phpcs`, indicating that it is valid. Now, if we wanted to run it against our entire directory, all we need to do is to point it the `source` directory where our files are. However, doing this and seeing errors for every file in a large directory could be really hard to read through.

To help with this, PHP_CodeSniffer also has a summary report function that can summarize each file and the number of errors and warnings found in each. It is invoked by passing in the `--report=summary` argument. As with running it directly against a valid file, if there are no issues, it will not be listed on the summary:

```
$ phpcs --report=summary --standard=PSR1,PSR2 source
PHP CODE SNIFFER REPORT SUMMARY
----------------------------------------------------------------------
FILE                                                  ERRORS  WARNINGS
----------------------------------------------------------------------
.../Apress/source/invalid.php             5       0
----------------------------------------------------------------------
A TOTAL OF 5 ERRORS AND 0 WARNINGS WERE FOUND IN 1 FILES
----------------------------------------------------------------------
```

PHP_CodeSniffer Configuration

There are a number of different configuration options and methods for configuring PHP_CodeSniffer. Going through all of them is out of the scope of this chapter, so the online documentation listed earlier is the best resource for finding all available options. However, let's look at a few different options and how we can set them.

■ **Note** PHP_codeSniffer can also run in batch mode if needed.

Default configurations can be changed using the `--config-set` argument. For example, to change the default standard to check against to be PSR-1 and PSR-2 rather than the PEAR standard that `phpcs` uses by default, it could be set this way:

```
$ phpcs --config-set default_standard PSR1,PSR2
Config value "default_standard" added successfully
```

You can also specify default configuration options directly in a project using a `phpcs.xml` file. This will be used if you run `phpcs` in a directory without any other arguments. Here is an example:

```
<?xml version="1.0"?>
<ruleset name="Apress_PhpDevTools">
    <description>The default phpcs configuration for Chapter 3.</description>

    <file>invalid.php</file>
    <file>valid.php</file>
```

```
    <arg name="report" value="summary"/>

    <rule ref="PSR1"/>
    <rule ref="PSR2"/>
</ruleset>
```

Within this file, the files to check are specified, as well as the rules we want to use. Multiple rules are specified using multiple `<rule />` tags.

PHP_CodeSniffer Custom Standard

In the event that you have your own standard, or have adopted most of PSR-1 and PSR-2 but decided to deviate from a rule here or there, you can create your own custom standard for `phpcs` to use. It is based off of the PSR-1 and PSR-2 standard and just overrides the parts that you wish to deviate from. This is done using a `ruleset.xml` file, which is then used with `phpcs` using the `--standard` argument, just as with any other coding standard.

At the very least, a `ruleset.xml` file has a name and a description and is formatted just as the `phpcs.xml` file we created is. However, just having the name and description does not provide `phpcs` with any instructions to override from an existing ruleset. For this example, say we want to change the standard to not restrict method names to `camelCase`. This would be done with a configuration like this:

```
<?xml version="1.0"?>
<ruleset name="Apress PhpDevTools CustomStandard">
    <description>A custom standard based on PSR-1 and PSR-2</description>

    <!-- Don't restrict method names to camelCase -->
    <rule ref="PSR1">
        <exclude name="PSR1.Methods.CamelCapsMethodName"/>
    </rule>

    <!-- Additionally, include the PSR-2 rulesets -->
    <rule ref="PSR2"/>

</ruleset>
```

With this ruleset, we see that all we needed to do was define a name for our rule, include the rulesets we wanted to base our standard off of, then specify the rule we wanted to exclude out of those rulesets. Now, if we run the validation against our `invalid.php` file we'll see the errors drop to four from five, as the method name violation is gone because our new standard doesn't restrict it to `camelCase`:

```
$ phpcs --standard=custom_ruleset.xml invalid.php

FILE: /Apress/source/invalid.php
----------------------------------------------------------------------
FOUND 4 ERRORS AFFECTING 3 LINES
----------------------------------------------------------------------
 3 | ERROR | Each class must be in a namespace of at least one level
   |       | (a top-level vendor name)
 5 | ERROR | Class constants must be uppercase; expected VERSION but
   |       | found version
 7 | ERROR | The var keyword must not be used to declare a property
 7 | ERROR | Visibility must be declared on property "$Property"
----------------------------------------------------------------------
```

PHP_CodeSniffer IDE Integration

As mentioned earlier, some IDEs such as PHPStorm and NetBeans have ways to integrate PHP_CodeSniffer directly within them. The exact process of configuring it for these IDEs can change as their respective vendors release new versions, so we won't cover this here. As of the time of this writing, the steps to set this up for PHPStorm are covered in the online documentation.

In my PHPStorm install, I have PHP_CodeSniffer configured and set to the PSR-1 and PSR-2 standards. With this configured, I get immediate feedback from PHPStorm if any of the code I'm writing deviates from these standards by way of a yellow squiggly line under the line of code that's in violation, as seen in Figure 3-1.

Figure 3-1. *Real-time PSR violation detection and hits in PHPStorm*

You can also run validation on the file and see the inspection rules directly within PHPStorm, as seen in Figure 3-2.

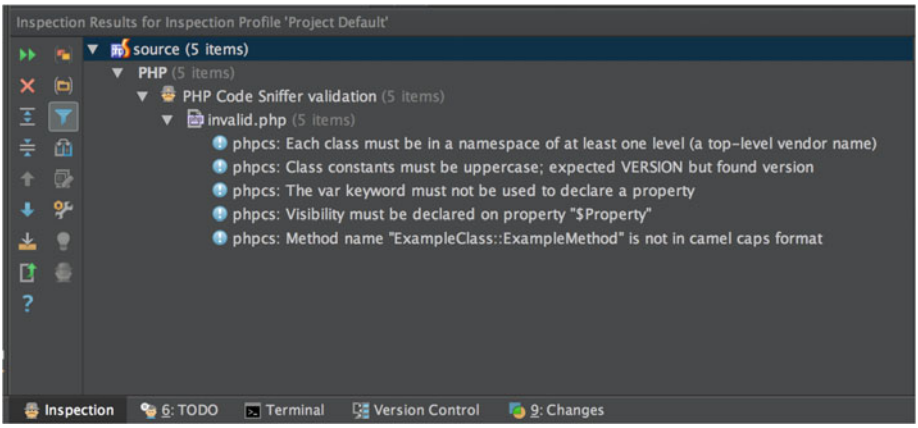

Figure 3-2. *PHP_CodeSniffer validation results within PHPStorm*

Code Documentation Using phpDocumentor

Not all coding standards provide rules regarding code comments. For example, PSR-1 and PSR-2 do not have set comment rules in place. However, it is just as important to establish a standard that everyone working on your project will follow when it comes to comments.

Arguably, one of the most popular formats for PHP is the DocBlock format used in conjunction with providing information about a class, method, function, or other structural element. When used in conjunction with the phpDocumentor project, you have the ability to automatically generate code documentation for your entire project and provide an easy reference for all developers to follow.

Another oft-used code documentation tool is PHPXref (`phpxref.sourceforge.net`). In general, PHPDocumentor and PHPXref have mainly two different targets:

- phpDocumentor is mainly used for generating real documentation from the source in a variety of different formats.
- PHPXref tool is mainly used to help the developer browse the code documentation of large PHP projects.

Installing phpDocumentor

phpDocumentor can be installed a few different ways. You can download the Phar file and execute it directly, you can install using Pear, or you can install using Composer. Here are the steps for each of these installation methods.

- First of all, you want to check the PEAR prerequisites:

  ```
  http://pear.php.net/manual/en/installation.php
  ```

- Download and execute the Phar files:

```
$ curl -OL http://www.phpdoc.org/phpDocumentor.phar
```

- If you have Pear installed you can install it using the PEAR installer. This is done with the following commands:

```
$ pear channel-discover pear.phpdoc.org
$ pear install phpdoc/phpDocumentor
```

- Lastly, you can use Composer to install it system wide with the following command:

```
$ composer global require "phpdocumentor/phpdocumentor:2.*"
```

Using phpDocumentor

As previously mentioned, phpDocumentor should be used to document structural elements in your code. phpDocumentor recognizes the following structural elements:

- Functions
- Constants
- Classes
- Interfaces
- Traits
- Class constants
- Properties
- Methods

To create a DocBlock for any of these elements, you must always format them the exact same way–they will always precede the element, you will always have one block per element, and no other comments should fall between the DocBlock and the element start.

A DocBlocks' format is always enclosed in a comment type called DocComment. The DocComment starts with `/**` and ends with `*/`. Each line in between should start with a single asterisk (*). The following is an example of a DocBlock for the example class we created earlier:

```
/**
 * Class ExampleClass
 *
 * This is an example of a class that is PSR-1 and PSR-2 compliant. Its only
 * function is to provide an example of how a class and its various properties
 * and methods should be formatted.
 *
```

```
 * @package Apress\PhpDevTools
 */
class ExampleClass
{
    const VERSION = '1.0';

    public $exampleProp;

    public function exampleMethod()
    {
        $this->$exampleProp = true;
    }
}
```

As we can see with this example, a DocBlock is broken into three different sections:

- Summary – The summary line should be a single line if at all possible and is just that–a quick summary of what the element is that we're documenting.
- Description – The description is more in-depth in describing information that would be helpful to know about our element. Background information or other textual references should be included here, if they are available and/or needed. The description area can also make use of the Markdown markup language to stylize text and provide lists and even code examples.
- Tags / Annotations – Lastly, the tags and annotations section provides a place to provide useful, uniform meta-information about our element. All tags and annotations start with an "at" sign (@), and each is on its own line. Popular tags include the parameters available on a method or function, the return type, or even the author of the element. In the preceding example, we use the package tag to document the package our class is part of.

■ **Note** Each part of a DocBlock is optional; however, a description cannot exist without a summary line.

Let's expand on the preceding example and provide DocBlocks for each of the structural elements of our example class:

```
<?php

namespace Apress\PhpDevTools;

/**
 * Class ExampleClass
```

```
 *
 * This is an example of a class that is PSR-1 and PSR-2 compliant. Its only
 * function is to provide an example of how a class and its various properties
 * and methods should be formatted.
 *
 * @package Apress\PhpDevTools
 * @author Chad Russell <chad@intuitivereason.com>
 */
class ExampleClass
{
    /**
     * Class version constant
     */
    const VERSION = '1.0';

    /**
     * Class example property
     *
     * @var $exampleProp
     */
    public $exampleProp;

    /**
     * Class example method
     *
     * This method is used to show as an example how to format a method that is
     * PSR-1 & PSR-2 compliant.
     *
     * @param bool $value This is used to set our example property.
     */
    public function exampleMethod($value)
    {
        $this->$exampleProp = $value;
    }

    /**
     * Gets the version of our class
     *
     * @return string Version number
     */
    public function classVersion()
    {
        return self::VERSION;
    }
}
```

Now that we've expanded our example, you can see several unique tags being used as well as a sampling of how you can mix the three sections together as needed. For a full listing of the tags that are available for phpDocumentor to use, see the full phpDocumentor online documentation at `http://www.phpdoc.org/docs/latest/index.html`.

Running phpDocumentor

In addition to having nice, uniform code comments for your project when using phpDocumentor and the DocBlock format, you also now have the power and ability to effortlessly generate code documentation that will transform your comments into a documentation resource. Once you have phpDocumentor installed, it's simply a matter of running it to produce this documentation.

There are just two of three command line options needed to produce your first set of documentation. The options are:

- `-d` – This specifies the directory or directories of your project that you want to document.
- `-f` – This specifies the file or files in your project that you want to document.
- `-t` – This specifies the target location where your documentation will be generated and saved.

For this example, we'll run it against our one example class from before:

```
$ phpdoc -f valid.php -t doc
```

Here, we're telling phpDocumentor to run against the file `valid.php` and to save the documentation in a new folder called `doc`. If we look in the new `doc` folder, we will see many different folders and assets required for the new documentation. You can view it by opening the `index.html` file, which is generated in a web browser. We can see what the page for our Example Class looks like in Figure 3-3.

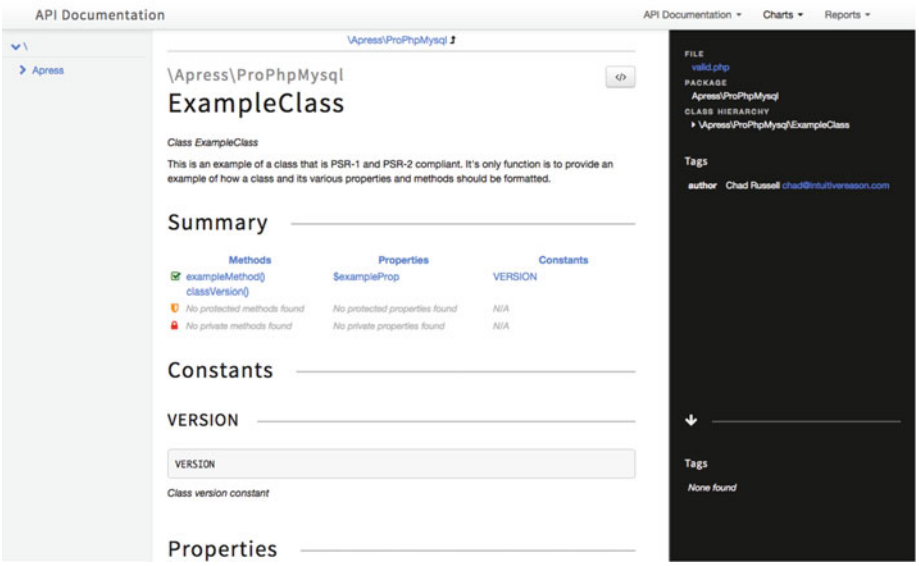

Figure 3-3. *phpDocumentor-generated class documentation*

Non-structural Comments

Lastly, since phpDocumentor only has provisions for structural comments, it is recommended that you establish guidelines for your coding standard that extend to non-structural comments. The Pear coding standard, for example, provides a general rule of thumb that is a great strategy to follow. Under their recommendations, you should always comment any section of code that you don't want to have to describe or whose functionality you might forget if you have to come back to it at a later date.

It's recommended that you use either the C-style comments (`/* */`) or C++ comments (`//`). It's discouraged to use the Perl/Shell-style comments (#), even though it is supported by PHP.

Summary

In this chapter, we discussed the benefits of using a coding standard for your projects. We took an in-depth look at the PHP-FIG PHP Standard Recommendations as well as some examples of code that follows these standards. We covered using the PHP_CodeSniffer tool as a validator of your code to ensure you and your team members are following your decided-upon standards. Lastly, we covered code commenting and documentation using the phpDocumentor project and the DocBlock format.

In the next chapter we will discuss Frameworks.