**The Single Responsibility Principle**

A function is easier to use, more reliable when reused, and easier to test when it does only one thing – that is, when it has a single responsibility. When you need another task to be performed, do not add it to your existing function; just write another one instead.

The syntax of a function is as follows:

function [identifier] ([[typeHint][…] [&]$parameter1[…][= defaultValue]][, [&]$p2, ..$pn]])[: [?]returnType|void]

{

// function body, any number of statements

[global $someVariable [, $andAnother]] // bad idea, but possible

[return something;]

}

Don't be put off by this apparently complex syntax definition. Functions are really easy to write, as the following examples in this chapter will show you.

However, let's now spend some time trying to break this syntax apart.

**The function Keyword**

The **function** keyword tells the PHP parser that what comes next is a function.

**Identifier**

The identifier represents the name of the function. Here, the general rules for identifiers in PHP will apply. The most important ones to remember are that it cannot start with a number and it cannot contain spaces. It can contain Unicode characters, although this is relatively uncommon. It is, however, quite common to define special, frequently used functions with underscores:

function \_\_( $text, $domain = 'default' ) {

return translate( $text, $domain );

}

This function is used to translate the text in WordPress templates. The idea behind it is that you will spot immediately that this function is something special and you won't be tempted to write a function with the same name yourself. It is also very easy to type, which is handy for frequently used functions. As you can see, it takes a required parameter, **$text**, to be translated. It also takes an optional **$domain**, in which the translation is defined, which is the **default** domain by default (a **text** domain in translations serves to separate different fields of interest that might have the same word for different things, so that these words can be translated differently if the other language has different words depending on the context). **\_\_ function** is what we call a wrapper for the translate function. It passes its arguments on to the **translate** function and returns the return value of the **translate** function. It is faster to type and it takes up less space in templates, making them more readable.

**Type Hints**

In the function declaration, type hinting is used to specify the expected data type of an argument. Type hints for objects have existed since PHP 5.0 and for arrays since PHP 5.1. Type hints for scalars have existed since PHP 7.0. Nullable type hints have existed since PHP 7.1. A type hint for **object** has existed since PHP 7.2. Consider the following example:

function createOutOfCreditsWarning(int $maxCredits, string $period, int $waitDays): string

{

$format = 'You have used the maximum amount of %d credits you are

allowed to spend in a %s. You will have to wait %d days before

new credits become available.';

return sprintf($format, $maxCredits, $period, $waitDays);

}

In the preceding example, there are three type hints. The first one hints that **$maxCredits** should be an integer. The second one hints that **$period** should be a string, and the third one hints that **$waitDays** must be an integer.

If a type hint is prefixed with a question mark, as in **?int**, this indicates that the argument must either be the hinted type or **null**. In this case, the hinted type is **integer**. This has been possible since PHP 7.1.

**The Spread Operator (…) with Type Hints**

The spread operator (**…**) is optional and indicates that the parameter has to be an array that only contains elements of the hinted type. Although it has existed since PHP 5.6, it is a rarely used yet very powerful and useful feature that makes your code more robust and reliable, with less code. There is no longer a need to check every element of a homogeneous array. When you define a parameter with such a type hint, you also need to call the function with the parameter prefixed with the spread operator.

The following is an example of a fictional function that I made up to demonstrate the use of the spread operator. The **processDocuments** function transforms XML documents using **eXtensible Stylesheet Language Transformations** (**XSLT**). While this is really interesting when you need to transform documents, it doesn't really matter for the demonstration of the spread operator. The spread operator is the three dots before **$xmlDocuments** in the function signature. It means that **$xmlDocuments** must be an array that contains only objects of the **DomDocument** hinted type. A **DomDocument** hinted type is an object that can load and hold XML. It can be processed by an object of the **XsltProcessor** class, to transform the document into another document. **XsltProcessor** in PHP is very powerful and very performant. You can even use PHP functions inside your XSL style sheets. This nifty feature should be used with caution, however, because it will render your XSL style sheets useless to other processors as they do not know PHP.

The return type of the function is **Generator**. This is caused by the **yield** statement inside the **foreach** loop. The **yield** statement causes the function to return each value (a document, in our case) as soon as it becomes available. This means it is efficient with memory: it does not keep the objects in memory in an array to return them all at once, but instead returns them one by one immediately after creation. This makes a generator very performant on large sets while also using fewer memory resources:

function processDocuments(DomDocument … $xmlDocuments):Generator

{

$xsltProcessor = new XsltProcessor();

$xsltProcessor->loadStylesheet('style.xslt');

foreach($xmlDocuments as $document){

yield $xsltProcessor->process($document);

}

}

The preceding function may appear pretty confusing, but it is fairly simple. Let's start with the usage of the spread operator; this is used to signify that the parameter will be required as an array. Additionally, the parameters are type hinted as **DomDocument**objects, meaning that the parameters will be an array of **DomDocument** objects. Moving onto the function, we define a new instance of **XsltProcessor** and load in a style sheet for the processor. Note that this is a conceptual example and more information on **XsltProcessor** and style sheets can be found in the PHP documentation at [php.net](http://php.net). Finally, we use a **foreach** loop to iterate through the array of documents and yield the results of the process method on each document. As document processing can be memory intensive, the use case for a generator is apparent if you can imagine passing a large array of documents to this function.

To call this function, use the following code:

// create two documents and load an XML file in each of them

$document1 = new DomDocument();

$document1->load($pathToXmlFile1);

$document2 = new DomDocument();

$document2->load($pathToXmlFile2);

// group the documents in an array

$documents = [$document1, $document2];

// feed the documents to our function

$processedDocuments = processDocuments(…$documents);

// because the result is a Generator, you could also loop over the

// result:

foreach(processDocuments(…$documents) as $transformedDocument) {

// .. do something with it

}

**Parameters in User-Defined Functions**

When defining a function, you are allowed to define parameters for it. When you are defining a parameter, consider whether it is expected to always be of the same type or whether you can force the developer using your code to always pass the same type. For example, when integer values are expected, a type hint of **int** is a good idea. Even if a developer passes **2**, which is a string, they can easily be educated to cast this to an integer before passing it to your function, using **(int) "2"**. More realistically, **2** would be stored in a variable. So, now you have a type hint:

int

Next, you should come up with a good name for your parameter. Ideally, it should be descriptive, but not overly long. When you expect a **DomDocument** code, **$domDocument**, **$xmlDocument**, or simply **$document** can be fine names, while **$doc** might be a little too short and confusing to some people and just **$d** would be just bad:

int $offset

Does a default value make sense for **$offset**? In most cases, it will be **0**, because we usually start a process at the beginning of something. So, **0** would make a great default value, in this case:

int $offset = 0

Now we have a parameter with a type hint of **int** and a default of **0**. The parameter is now optional and should be defined after the parameters that are not optional.

If a parameter cannot be expected to always be of the same type, processing it in your function may be harder, because you might have to check its type in order to decide how you should treat it. This makes unit testing your function harder and it complicates fault finding if things go wrong, since your code will have several paths of execution, depending on the type of input.

When a parameter is prefixed with **&**, it means that if a scalar is passed, it will be passed by reference, instead of as a copy or literal. Objects are always passed by reference and, therefore, using **&** on an object parameter is redundant and does not change the behavior of the function.

**Return Types in User-Defined Functions**

Return types are written as a colon followed by the type name. Return types were introduced in PHP 7. They make your code more robust because you are more explicit about what you expect from your function, and this can be checked at compile time rather than failing at runtime when something goes wrong, possibly in production. If you use an IDE, it will warn you when a return type does not match what you actually return or expect from the function. This means you can correct the error before it hits your users.

In the preceding example, the **processDocuments** function has a return type of **Generator**. A **Generator** type generates values and makes them available as soon as possible. This can be very performant: you don't have to wait for all the values to become available before processing them further. You can start with further processing as soon as the first value comes out of the **Generator** type. The **Generator** type churns out a value each time the **yield** language construct is used.

**yield** was introduced in PHP 5. At the time of writing, we are at PHP 7.3 and there are still many developers who have never used **yield** or do not even know what it does. When you are processing arrays or records from a database, for example, and you need extreme performance, consider whether you have a use case for a **Generator** type.

You can use **void** as the return type to indicate that nothing is returned from the function.

**Signature**

The following part of the function declaration is called the **signature**:

([typeHint [&]$parameter1[= defaultValue], [&]$p2, …])[: returnType]

So, the signature of a function defines its parameters and the return type.

**Returning a Value**

A function may return a value or not. When the function does not return anything, not even **null**, the return type can be void as of PHP 7.1. Values are returned by typing **return** followed by what you want to return. This can be any valid expression or just a single variable or literal:

return true;

return 1 < $var;

return 42;

return $documents;

return; // return type will be "void" if specified

return null; // return type must be nullable if specified

**Parameters and Arguments**

Functions accept arguments. An argument is a literal, variable, object, or even callable that you pass into a function for the function to act upon. If a parameter is defined at the position of the argument, you can use the argument inside your function by using the name of the parameter. The number of parameters may be variable or fixed. *PHP allows you to pass more parameters than the function signature defines*. If you want dynamic parameters, PHP has two built-in functions that make this possible; you can get the number of parameters with **func\_num\_args()** and the parameters themselves with **func\_get\_args()**. To show these functions in action, I will give you an example.

Here's an example of using **func\_num\_args()**. In this example, we define a method that will have no predefined parameters/arguments. But using the built-in **func\_num\_args**function, we will be able to count how many parameters/arguments are passed:

function argCounter() {

$numOfArgs = func\_num\_args();

echo "You passed $numOfArgs arg(s)";

}

argCounter(1,2,3,4,5);

The output is as follows:

You passed 5 arg(s)

Now that we can count the number of arguments, we can combine that function with **func\_get\_args()** to loop through and see what was passed. Here's an example of using **func\_get\_args()**:

function dynamicArgs(){

$count = func\_num\_args();

$arguments = func\_get\_args();

if($count > 0){

for($i = 0; $i < $count; $i++){

echo "Argument $i: $arguments[$i]";

echo PHP\_EOL;

}

}

}

dynamicArgs(1,2,3,4,5);

The output is as follows:

Argument 0: 1

Argument 1: 2

Argument 2: 3

Argument 3: 4

Argument 4: 5

**Optional Parameters**

Parameters to functions are optional when they have default values defined for them:

function sayHello($name = 'John') {

return "Hello $name";

}

This function defines a parameter, **$name**, with a default value of **John**. This means that when calling the function, you do not need to provide the **$name** parameter. We say that the **$name** parameter is optional. If you do not provide a **$name** parameter, **John** will be passed anyway for the **$name** parameter. Optional parameters should be defined at the very end in the function signature, because, otherwise, if any required parameters come after the optional ones, you would still have to provide the optional parameters when calling the function.

The example is in **function-with-default-value.php**. The various usages are documented in the **TestSayHello.php** unit test.

**Parameters Passed by Reference to Our Function**

Remember the **countMe** function? It used a global variable named **$count** to keep track of how many times the function was called. This could also have been accomplished by passing the **$count** variable by reference, which is also a slightly better practice than polluting the global scope from within your function:

<?php

function countMeByReference(int &$count): void

{

$count++;

}

Use it further down in the same script, as follows:

$count = 0;

countMeByReference($count);

countMeByReference($count);

countMeByReference($count);

echo $count; // will print 3

Please note that calling methods in the same script as they are defined in is perfect for exercises and playing with code and also for simple scripts, but doing this is actually a violation of PSR-1. This is a coding convention that states that files either define functions (not causing side effects) or use them (causing side effects).

**Default Values for Parameters**

In the following example, we are demonstrating the use of default values. By defining a default value, you give the developer using the function the ability to use the function as is without having to pass their own value.

Consider the following example:

/\*\*

\* @param string $systemTempDirectory

\* @return string

\*/

function determineOutputDirectory(string $systemTempDirectory = '/tmp'): string

{

return $systemTempDirectory . DIRECTORY\_SEPARATOR . 'output';

}

Between the parentheses is the function signature, which consists of a single parameter, **$systemTempDirectory**, with a type hint of **string** and a default value of **/tmp**. This means that if you pass a directory with your function call, it must be a string. If you do not pass an argument, the default value will be used.

COMPLETE & CONTINUE