PHP 8 in a Nutshell

Amit D. Merchant

amitmerchant.com

Table of Contents

Introduction **4**

PHP 8 ..... **5**

Union Types 6

The Nullsafe Operator 9

Constructor Property Promotion 11

New String functions 14

Non-capturing Exception Catches 18

The Mixed Type 19

Match Expressions 25

Named Arguments 29

Attributes 32

The ::class keyword 39

Throw as an Expression 41

The get_debug_type() function 43

PHP 8.1 ..... **46**

Readonly Properties 47

Native Enumerations 51

Fibers or Coroutines 55

Intersection Types 58

First-class callables 60

New in initializers 63

Array unpacking with string keys 66

The array_is_list() function 68

The Never Return Type 71

Final class constants 74

Conclusion **76**

Named Arguments

Did you ever get in a situation where you're seeing a function and its parameters and wonders what those parameters are all about? I'm pretty sure you did.

For instance, take following,

```
array_slice($array, $offset, $length, true);
```

Now, the first three parameters passed to array_slice is seemed to be self-explanatory because of informative variable names but what about the fourth parameter? It just says `true`. But what does `true` signify here? Well, to find it out, you'd need to navigate to the function definition or refer to the documentation. In the case of `array_slice`, the definition of it is like so.

```
function array_slice(
    array $array,
    $offset,
    $length = null,
    $preserve_keys = false
) { }
```

So, the fourth parameter is `$preserve_keys` which you can tell by looking at the definition. But you don't have anything using which you can tell just by looking at the function declaration.

Well, this can be solved by using the newly added **"named parameters"** in PHP 8.

Named parameters

In a nutshell, named arguments/parameters allow passing arguments to a function based on the parameter name using the following syntax.

```
callAFunction(paramName: $value);
```

Here, named arguments are passed by prefixing the value with the parameter name followed by a colon.

So, the previous example of `array_slice` can be written using named parameters like so.

```
array_slice($array, $offset, $length, preserve_keys: true);
```

As you can see, the code is now pretty self-documenting as compared to the previous example. Now, we know what the fourth param is intended to do.

Apart from this, this would be very convenient when a function has a high number of parameters or default ones. With named arguments, we can make the code much more readable.

Order-agnostic paramters

Apart from making the code self-documenting, this feature also makes the parameters *order agnostic*, and allows skipping default values arbitrarily.

Meaning, the order of the arguments is not important when using named arguments.

To give you a simple example:

```
// Using positional arguments:
array_fill(0, 100, 50);

// Using named arguments:
array_fill(start_index: 0, num: 100, value: 50);
```

The order in which the named arguments are passed does not matter. The above example passes them in the same order as they are declared in the function signature, but any other order is possible too:

```
array_fill(value: 50, num: 100, start_index: 0);
```

As you can tell, the order of the arguments have been entirely changed and yet, nothing breaks!

Skipping defaults

With named parameters, it's possible to not specify all the defaults until the one you want to change. Named arguments allow you to directly overwrite only those defaults that you wish to change.

So, the following would be perfectly fine...

```
htmlspecialchars($string, double_encode: false);
```

Instead of specifying all the default values like so...

```
htmlspecialchars($string, default, default, false);
```

Pretty cool, no?

In closing

We're just scratching the surface here. You can refer to the original RFC if you want to take a more in-depth look at named parameters and constraints they comes with.