What's new in PHP 8

Free ebook, first distributed during Laravel-Nagpur meetup on November 27th, 2020.

Latest version of this ebook can be freely downloaded from https://github.com/kapilsharma/whatsnewinphp8

Kapil Sharma

Table of Contents

Introduction

Book version 1.0.0

This book is just the notes made by the author (Kapil Sharma) while going through new features of PHP 8. I'll be sharing this as an ebook, in case someone finds these notes helpful.

I'll be first sharing this book during Laravel-Nagpur meetup on November 27th, 2020 at 7:30 pm IST, presented by me and live-streamed on YouTube. If you are reading the book later, you may see the recording on that meetup on Laravel-Nagpur's YouTube channel (https://bit.ly/laravel-ngp-nov-2020)

PHP 8 is the next major version of PHP. As like in any major version, there will be some breaking changes. That means we can not update PHP version on the server and expect everything to be working.

We are dividing this ebook into few sections:

- Major new changes.
- New functions.
- Syntax changes
- Breaking changes
- Internal improvements

Code examples.

During the meet-up, we will be using a practical approach, lot of examples will be shown during the meet-up and also used in this book. You can get all of these examples by cloning Git repository (https://github.com/kapilsharma/WhatsNewInPHP8).

Once you clone the repository, it has the following folders:

- `assets` - Ignore them. They are some assets used for making the ebook.
- `contents` - You can ignore them as well. It contains markdown files used to generate ebook. You may better use PDF ebook. If you wish to make any correction in the generated PDF, you may consider to fix the error/typo in related markdown and probably send back pull request with the correction.
- `example` - This folder contains the example PHP code used in the ebook. Each feature given in this book, have its folder under example and contains the code of examples given in the section. Even in the ebook, the file name of the example is there above

the code.
- export - You will not get this folder after clone. However, if you decide to generate the ebook, the folder will be created automatically. Thanks to Mohamed Said, I'm using ibis to generate this ebook. If you wish to make some correction (please consider PR) and regenerate the ebook yourself, just setup ibis and run ibis build command. This folder is added to '.gitignore' file to avoid committing it.

Running the examples in PHP 8

At the time of writing this ebook, PHP 8 is still not released. I'm pretty sure it will be released by the time you will be reading this ebook. If not, or you may not install PHP 8 on your system for some reasons, you may use PHP 8 docker with following command (Expecting docker is installed on your system, if no, just DuckDuckGo as I can't cover docker installation in this book):

```
docker run -it -v "$PWD":/usr/src/app -w /usr/src/app php:8.0.0RC5
/bin/bash
```

Please note, `PHP 8.0.0 Release candidate 5` is latest at the time of writing this ebook. You may want to check PHP's official dockerhub (https://hub.docker.com/_/php?tab=tags) to check the latest PHP 8 version available and change `php:8.0.0RC5` in the above command with the latest version.

Also, it is recommended to run `composer install` (locally, not in docker) to generate required autoloading, which is needed for few examples.

License

You can use these notes as you wish for personal learning. However, if you wish them to improve and share, please consider sending a pull request for improvement and keep the original author's name. In short, if you want to use these notes in any way, feel free to use it under Creative Commons Attribution Share Alike 4.0 International license.

About author

Short intro: I'm Kapil Sharma, developing web applications since 15+ years in different technologies including PHP. I started my career in 2004, as a web designer, working in HTML 4 (Without CSS, using huge tables with a lot of rowspan and colspan for designing web pages). Soon I started using CSS 2, JavaScript and PHP 4/5. Later I started working as a developer in PHP and Java/J2EE. By the end of 2005, I completely started working on Java/J2EE and worked on it for next 3 years, mostly on OFBiz, Mule, Service Mix (Java) and occasionally on Joomla version 1 (PHP). In 2009, I worked on PHP, Python and Ruby on Rails and since 2010, I started working mostly in PHP.

I used PHP without using any framework, Code Igniter, Zend Framework 1, Symfony and nowadays mostly Laravel and Node JS along with Angular and Vue JS for frontend.

I also help to manage PHP Reboot, a developers' community in Pune, India since 2014 and a regular speaker on PHP Reboot and other meetups.

Right now, I'm working as Technical Architect at Cactus Global, Mumbai but I'm serving my notice period and soon will be joining Parker Consulting in Pune.

Credits

While learning new features of PHP 8, making my notes and converting them into this ebook, I went through a lot of websites, articles, tutorials, PHP manual and official RFC. I mostly used search engine (Google and DuckDuckGo) to find them. I did not remember the link of all those pages but I'm thankful to everyone who posted any article/tutorial about new features of PHP 8, they helped me learn new features of PHP 8, make my notes and this book.

Most important were the PHP RFC, which I added as a source wherever possible. Until PHP 8 documentation is available, RFC is the best place to understand an upcoming feature.

I'd like to specially mention stitcher.io blogs, which help me to understand a few RFC in details.

I used free service of canva.com to design the cover page of the book and would like to thanks them for providing an online tool to create a free book cover.

Issues

If you find any issue in the book, feel free to send a PR or at least create a bug report (issue at https://github.com/kapilsharma/WhatsNewInPHP8/issues) on GitHub. I will frequently check the issue and fix the issues in next version of the book.

Improvements, book version

There is a chance of improvement in every work. I'll be doing it myself and fixing the issue raised by others. Make sure you have latest version of the book. Book version is given at the top of the introduction section of the book and on GitHub (https://github.com/kapilsharma/whatsnewinphp8). If you find you have older version, please download the newer version of the book with bug fix and new features.

Major new changes

In this section, we will discuss new features introduced in PHP version 8.

It is not a book to learn PHP but Kapil's notes while going through new PHP 8 features, converted into a form of EBook. I assume you already know PHP and its features available till PHP version 7.4. Thus, without going into much theory, let's dive directly into changes and their examples.

Please note, in some sections, we will also discuss the history to understand features that came in older versions. It could be useful for those who are not aware or remember changes in PHP 7 or earlier versions. If you wish to skip the history, go to the title `Changes in PHP 8`.

Union types

RFC: https://wiki.php.net/rfc/union_types_v2

History

Skip to title `Changes in PHP 8` if you are interested to see only the changes made in PHP 8.

Dynamically typed language

PHP is a dynamically typed language, that means we can assign any data-type to any variable. Following code is perfectly fine in PHP:

File: examples/UnionTypes/DynamicTypes.php

```php
<?php

$intType = null; // Variable can be null, may be no problem.
echo "Value of variable is: $intType" . PHP_EOL;

$intType = 2; // Right and expected type.
echo "Value of variable is: $intType" . PHP_EOL;

$intType *= 1.4; // Wait, it will be 2.8 and PHP will cast it to a
float, doesn't match with name of variable.
echo "Value of variable is: $intType" . PHP_EOL;

$intType = "Let's make it string, I don't care for types"; // Perfectly
fine.
echo "Value of variable is: $intType" . PHP_EOL;
```

It runs perfectly fine with following output.

```
Value of variable is:
Value of variable is: 2
Value of variable is: 2.8
Value of variable is: Let's make it string, I don't care for types
```

Since PHP is still supposed to be a dynamically typed language, PHP 8 kept this behaviour so we will be having the same output even in PHP 8.

Although many languages support dynamic types as it is easy to manage, there is a downside that we may accidentally change a variable type and introduce a bug.

PHP started fixing this problem and in PHP 5, we can define type-hinting in functions/methods. However, it was limited only to class names. PHP 7 allowed to type hint for internal data types. Following example demonstrate that.

File: examples/UnionTypes/TypeHinting.php

```php
class TypeHinting
{
    public $intType;

    public function __construct(int $intType)
    {
        $this->intType = $intType;
    }

    public function printValue()
    {
        echo 'Value of variable is ' . $this->intType . PHP_EOL;
    }
}

$instance = new TypeHinting(2);
$instance->printValue();

$instance->intType = 2.2;
$instance->printValue();
```

As shown in constructor `__construct(int $intType)`, we defined parameter must be of type integer. Thus, if we pass any non-integer parameter to the constructor, it will throw an error and the developer will immediately know and fix it.

It still did not solve the problem as our instance variable `intType` is public that means, we can still assign any value to it directly. We exactly did that in the line `$instance->intType = 2.2;` and now, it is a float. The correct way is, instance variables must be private or protected and must be managed by functions. Although not recommended, in rare cases, we might need to make an instance variable public. PHP fix that problem in version 7.4 with typed properties (https://wiki.php.net/rfc/typed_properties_v2). With typed properties, we may rewrite the above example again.

File: examples/UnionTypes/TypeHinting2.php

```
class TypeHinting2
{
    public int $intType;

    public function __construct(int $intType)
    {
        $this->intType = $intType;
    }

    public function printValue()
    {
        echo 'Value of variable is ' . $this->intType . PHP_EOL;
    }
}

$instance = new TypeHinting2(2);
$instance->printValue();

$instance->intType = 2.2;
$instance->printValue();

$instance->intType = "3.5 is a float with some extra string";
$instance->printValue();

$instance->intType = "Now we have a string that can't be casted to int";
$instance->printValue();
```

Here, we changed `public $intType;` to `public int $intType;`. With this, we tell PHP that instance variable `$intType` may contain only integer values. Let's see the output of the program before discussing it.

```
Value of variable is 2

Value of variable is 2

Notice: A non well formed numeric value encountered in
/home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php on line 35
Value of variable is 3

PHP Fatal error:  Uncaught TypeError: Typed property
TypeHinting2::$intType must be int, string used in
/home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php:38
Stack trace:
#0 {main}
  thrown in /home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php on line 38
```

We first set an integer with `$instance = new TypeHinting2(2);` and output was as expected `Value of variable is 2`.

In next line, we set float value to the variable as `$instance->intType = 2.2;`. Here, PHP did the internal cast and made float to integer. Thus, output was even though not expected as human, it is expected `Value of variable is 2` (Not 2.2)

Next, we assign a string starting with a integer `$instance->intType = "3.5 is a float with some extra string";`. Now there is a problem but PHP still manage to cast it with notice but the program still executed and continued.

```
Notice: A non well formed numeric value encountered in
/home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php on line 35
Value of variable is 3
```

Important: In PHP 8, this will throw TypeError, not warning.

In the end, we push PHP to its limit and assigned a string `$instance->intType = "Now we have a string that can't be cast to int";`. Now PHP cannot auto-cast it to an integer and finally throw an error and stop executing program.

```
Fatal error: Uncaught TypeError: Typed property TypeHinting2::$intType
must be int, string used in
/home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php:38
Stack trace:
#0 {main}
  thrown in /home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/TypeHinting2.php on line 38
```

Even though PHP is dynamically typed language, with above changes, it was making things more strict, to make it easy to identify possible bugs during development time only.

Everything comes at a price, with type-hinting, we faced a new problem as PHP does not support Method-overloading. Assume we need to make add function, which should work on both integer and float. In strictly types language, which supports overloading with different signature (method name with parameter list), the following code should be perfect.

File: examples/UnionTypes/Overloading.php

```
Class Overloading
{
    public function add(int $number1, int $number2): int
    {
        return $number1 + $number2;
    }

    public function add(float $number1, float $number2): float
    {
        return $number1 + $number2;
    }
}

$instance = new Overloading();
echo '1 + 2 = ' . $instance->add(1, 2) . PHP_EOL;
echo '1.1 + 2.2 = ' . $instance->add(1.1, 2.2) . PHP_EOL;
```

However, PHP do not support method overloading thus above code throw following error

```
Fatal error: Cannot redeclare Overloading::add() in
/home/kapil/dev/ebooks/WhatsNewInPHP8/examples/101-
UnionTypes/Overloading.php on line 25
```

Although we can have a workaround of defining add with float and cast result to integer, it is not very convenient. Thus, PHP 8 comes with `Union Types`.

Changes in PHP 8

With `Union Types`, we can define multiple type hinting for a variable as follow:

File: examples/UnionTypes/Overloading.php

```
Class UnionTypes
{
    public function add(int|float $number1, int|float $number2):
int|float
    {
        return $number1 + $number2;
    }
}

$instance = new UnionTypes();
echo '1 + 2 = ' . $instance->add(1, 2) . PHP_EOL;
echo '1.1 + 2.2 = ' . $instance->add(1.1, 2.2) . PHP_EOL;
```

Output

```
1 + 2 = 3
1.1 + 2.2 = 3.3
```

As seen in the above example, we type hinted $number1, $number2 and return type as `int|float`, which says it could be either integer or float.

One important thing to note, PHP 7.1 introduced nullable types (https://www.php.net/manual/en/migration71.new-features.php), where we can define type as `?string` meant it is either String or null. It will not work with union types. However, we can achieve the same result with `string|null`.

Nullsafe operator

RFC: https://wiki.php.net/rfc/nullsafe_operator

History

Skip to title `Changes in PHP 8` if you are interested to see only the changes made in PHP 8.

PHP ternary operator

The ternary operator is available in PHP since very long thus hopefully, you already know about it. Its syntax is `(expr1) ? (expr2) : (expr3)`, it is equivalent to

```
if ($expr1) {
    return $expr2;
} else {
    return $expr3;
}
```

Ternary shortcut (?:)

The shortcut operator does not have `expr2` of the above example. It is like `(expr1) ?: (expr2)`. Let's understand it by an example:

```
$result = $variable1 ?: $variable2;
```

Which is equivalent to

```
$result = $variable1 ? $variable1 : $variable2;
```

or

```
if ($variable1) {
    $result = $variable1;
} else {
    $result = $variable2;
}
```

Null coalescing operator (??)

PHP 7.0 introduced `Null coalescing operator (??)`, which confuse some people between `??` and `?:` operators. Following example is taken from PHP manual (https://www.php.net/manual/en/migration70.new-features.php):

`The null coalescing operator (??) has been added as syntactic sugar for the common case of needing to use a ternary in conjunction with isset(). It returns its first operand if it exists and is not NULL; otherwise, it returns its second operand.`

```
<?php
// Fetches the value of $_GET['user'] and returns 'nobody'
// if it does not exist.
$username = $_GET['user'] ?? 'nobody';
// This is equivalent to:
$username = isset($_GET['user']) ? $_GET['user'] : 'nobody';

// Coalescing can be chained: this will return the first
// defined value out of $_GET['user'], $_POST['user'], and
// 'nobody'.
$username = $_GET['user'] ?? $_POST['user'] ?? 'nobody';
?>
```

To make it more clear, let's see a few more examples with a comparison between `??` and `?:`:

```
// Lets compare them on null variable.
$a = null;
print $a ?: 'b'; // Output: b, because if(null) returns false
print $a ?? 'b'; // Output: b, because isset(null) returns false

// Let's check a variable which is not defined
print $c ?: 'a'; // Output: Notice: Undefined variable: c
print $c ?? 'a'; // Output: a, because isset($c) is false.

// Let's check it once again with an array key
$b = array('a' => null);
print $b['a'] ?? 'd'; // Output: d, because $b['a'] is null, which is
not true.
print $b['a'] ?: 'd'; // Output: d, because isset($b['a']) = isset(null)
= false
```

Thus, ternary shortcut (?:) check the variable for truth (`if ($variable)`) and null coalescing operator check the variable with `isset()` method. You may check much better comparison between these two conditions on PHP Manual (https://www.php.net/manual/en/types.comparisons.php)

Changes in PHP 8: nullsafe operator (?->)

Problem with the null coalescing operator is, it works only with variables. Thus, PHP 8 introduced nullsafe operator (?->)(Check RFC: https://wiki.php.net/rfc/nullsafe_operator) to solve this problem.

Let's understand it with an example:

examples/NullsafeOperator/NoNullsafePhp7.php

```
class Address
{
    public $country;

    public function __construct()
    {
        // Adding dummy country for demonstration purpose
        $this->country = 'India';
    }
}
```

```php
class User
{
    protected $address;

    public function __construct()
    {
        // Adding dummy address for demonstration purpose
        $this->address = new Address();
    }

    public function getAddress()
    {
        return $this->address;
    }
}

class Session
{
    public $user;

    public function __construct()
    {
        // Adding dummy user for demonstration purpose
        $this->user = new User();
    }
}

$session = new Session();

$country = null;

if ($session !== null) {
    $user = $session->user;

    if ($user !== null) {
        $address = $user->getAddress();

        if ($address !== null) {
            $country = $address->country;
        }
    }
}

echo "Country is " . $country . PHP_EOL;
```

I've taken an example of original RFC and extended it to be executable on the command line. Although code is simple and self-explaining, let me explain it. Here, I try to demonstrate a web session, session contains a user, who has an address containing the user's country. As we need to validate if the required information is present, we need to have three if conditions:

```
if ($session !== null) {
    $user = $session->user;

    if ($user !== null) {
        $address = $user->getAddress();

        if ($address !== null) {
            $country = $address->country;
        }
    }
}
```

With nullsafe operator, this whole code can be written in one line as `$session?->user?->getAddress()?->country`. Complete code will be:

File: examples/NullsafeOperator/Nullsafe.php

```php
class Address
{
    public $country;

    public function __construct()
    {
        // Adding dummy country for demonstration purpose
        $this->country = 'India';
    }
}

class User
{
    protected $address;

    public function __construct()
    {
        // Adding dummy address for demonstration purpose
        $this->address = new Address();
    }

    public function getAddress()
    {
        return $this->address;
    }
}

class Session
{
    public $user;

    public function __construct()
    {
        // Adding dummy user for demonstration purpose
        $this->user = new User();
    }
}

$session = new Session();

$country = $session?->user?->getAddress()?->country;

echo "Country is " . $country . PHP_EOL;
```

Named arguments (also known as Named parameters)

RFC: https://wiki.php.net/rfc/named_params

(Taken from RFC) Named arguments allow passing arguments to a function based on the parameter name, rather than the parameter position. It makes the meaning of the argument self-documenting, make them order-independent, and allows skipping default values arbitrarily.

Let's check it with an example:

File: examples/NamedArguments/NamedArguments.php

```php
<?php
class NamedArguments
{
    public function printUserInfo(
        string $name,
        int $age,
        string $occupation,
    ) {
        echo "Hello $name, you are $age years old and work as
$occupation." . PHP_EOL;
    }
}

$instance = new NamedArguments();

$instance->printUserInfo(
    name: 'Kapil',
    age: 38,
    occupation: 'Developer'
);

$instance->printUserInfo(
    occupation: 'Student',
    age: 8,
    name: 'Pari'
);
```

As seen in examples, with named parameters, we have given a name of each parameter and the order of parameters is no longer important. It also helps better-documented parameters right in the method signature.

It will also be helpful in case when we have one or more optional arguments. Let's update our example with some optional parameters:

File: examples/NamedArguments/NamedArguments.php2

```
class NamedArguments2
{
    public function printUserInfo(
        string $name,
        int $age,
        string $occupation = '',
        string $company = '',
        string $school = ''
    ) {
        $output = "Hello $name, you are $age years old,";

        if ($occupation !== '') {
            $output .= " you are a $occupation";
        }

        if ($school !== '') {
            $output .= " and study in $school";
        }

        if ($company !== '') {
            $output .= " and work in $company";
        }

        echo $output . PHP_EOL;
    }
}

$instance = new NamedArguments2();

$instance->printUserInfo(
    name: 'Kapil',
    age: 38,
    occupation: 'Developer',
    company: 'Cactus Global'
);

$instance->printUserInfo(
    school: 'Some School',
    occupation: 'Student',
    age: 8,
    name: 'Pari'
);
```

As we can see in two function call, we can not only move the order of parameters, we also no longer need to provide the first optional argument if we need to give the second optional argument. If we have to call the same function in PHP 7, we would have to do

```
$instance->printUserInfo('Kapil', 38, 'Developer', 'Cactus Global');
$instance->printUserInfo('Pari', 8, 'Student', '', 'Some School');
```

In the second call, even though we do not need company, we still have to pass an empty string in PHP 7 and earlier version.

Attributes

RFC: https://wiki.php.net/rfc/attributes_v2

What is metadata?

In programming, metadata is a `data that defines another data`, on in other words `data about data`. Confusing, let's see an example:

```
/**
 * Returns user model by user id
 *
 * @param int $userId
 * @return User User Model fetched from user id.
 */
public function getUser($userId): User
{
    // Code to fetch and return user object.
}
```

You must have seen similar code, where we define some documentation in Docblock comment. Here, two important things are, `@param` and `@return`, what they are? It does not contribute anything to the code, but a document generator use them to generate documentation and IDE like PHP Storm will read this docblock to show a warning at the time of development. For example, if you write `$userService->getUser('Kapil')`, your IDE will show an error that `function expects int but we are passing a string`. This will happen even though we did not type-hint in method parameters but because IDE will do it by reading `@param`, where we defined `$userId` must be an integer.

Thus, these docblocks may define what our data is (data about the data) and known as metadata.

However, for PHP interpreter, this whole docblock is just a comment and will be ignored. In other words, there is no inbuilt support in PHP to define metadata. It will now change in PHP 8 with Attributes.

Attributes are also known as `annotations` in a few other languages. The attributes, in PHP 8 are supposed to add metadata to classes, methods, properties/variables, method parameters, etc. Let's check an example of where these properties can be used (Taken from RFC)

```
#[ExampleAttribute]
class Foo
{
    #[ExampleAttribute]
    public const FOO = 'foo';

    #[ExampleAttribute]
    public $x;

    #[ExampleAttribute]
    public function foo(#[ExampleAttribute] $bar) { }
}

$object = new #[ExampleAttribute] class () { };

#[ExampleAttribute]
function f1() { }

$f2 = #[ExampleAttribute] function () { };

$f3 = #[ExampleAttribute] fn () => 1;
```

Practical use of attributes

Let's see a test code to see the practical use of Attributes. We first need to define an attribute.

File: examples/Attributes/TestAttribute.php

```
#[\Attribute]
class TestAttribute
{
    public string $testArgument;

    public function __construct(string $testArgument)
    {
        $this->testArgument = $testArgument;
    }
}
```

Here, we made a new `TestAttribute` class. It's a `custom attribute` as we defined it for our project/example. To define a custom attribute, we use a `global attribute`, which is `#[\Attribute]`. Custome attribute in our example is nothing but just a PHP class, attributed by `#[\Attribute]` and can store some extra data (`$testArgument`).

Now we defined a custom attribute, let's use it in a class.

File: examples/Attributes/TestClass.php

```
#[TestAttribute('Some metadata for TestClass')]
class TestClass
{

}
```

Here we defined a TestClass and added our custom attribute (TestAttribute) with some more metadata (Some metadata for TestClass).

Now we are set to fetch this metadata through code.

File: examples/Attributes/testing_attributes.php

```
$reflection = new \ReflectionClass(TestClass::class);
$classAttributes = $reflection->getAttributes();

echo $classAttributes[0]->newInstance()->testArgument . PHP_EOL;
```

Its output will be `Some metadata for TestClass`, which is our actual metadata. In our code, first line creates a `ReflectionClass` of `TestClass`. Later, we get the attributes of the class (TestClass) but `getAttributes` method, which will be an array. Since we have only one attribute, we are using the first array element, creating an instance of the class and fetching `testArgument`, which we defined while making the class.

How it will impact me as a developer?

Well, for the majority of developers, it will not, at least not directly and immediately.

It will be used by IDE to fetch metadata in a standard way provided we define it. Also, after some time, newer versions of frameworks like Laravel, Symfony, Zend Framework, Yii, Cake PHP, Drupal, Magento etc will be using attributes to define a lot of things. Symfony already declared in a blog post(https://symfony.com/blog/new-in-symfony-5-2-php-8-attributes) that starting version 5.2, they will be using attributes to define routes.

It is good to know about them so that they do not seem alien when first introduced in your favourite framework.

Constructor property promotion

RFC: https://wiki.php.net/rfc/constructor_promotion

We have a similar thing in JavaScript, which allow writing less code, achieving the same result.

Problem

(Taken from RFC) Currently, the definition of simple value objects requires a lot of boilerplate, because all properties need to be repeated at least four times. Consider the following simple class:

```php
class Point {
    public float $x;
    public float $y;
    public float $z;

    public function __construct(
        float $x = 0.0,
        float $y = 0.0,
        float $z = 0.0,
    ) {
        $this->x = $x;
        $this->y = $y;
        $this->z = $z;
    }
}
```

The properties are repeated:

1. In the property declaration,
2. The constructor parameters, and
3. two times in the property assignment.

Additionally, the property type also repeated twice.

Especially for value objects, which commonly do not contain anything more than property declarations and a constructor, this results in a lot of boilerplate and makes changes more complicated and error-prone.

Solution

PHP 8 will now support `Constructor Property Promotion`. With this, the above code can be written as:

File: examples/ConstructorPromotion/Point.php

```
class Point
{
    public function __construct(
        public float $x = 0.0,
        protected float $y = 0.0,
        private float $z = 0.0,
    ) {}

    public function print()
    {
        echo "Point($this->x, $this->y, $this->z)" . PHP_EOL;
    }
}

$point = new Point(1, 2.2);
$point->print();
```

Pretty, simple and a lot of fewer lines to achieve the same result, right?

The output of the above program is

```
Point(1, 2.2, 0)
```

In simple words, if we define access modifier (public, protected, private) in the class constructor, that variable will behave as an instance variable and we need not declare it separately as an instance variable.

Static return type

RFC: https://wiki.php.net/rfc/static_return_type

Before we can understand `static return type`, we must first understand `Late static binding`, introduced in PHP 5.2. You are already aware of that, skip next section and go to `Static return type`.

Late static binding

In PHP, if we want to return an instance of the same class, you must have used/seen `return new self` or `return new static`. What is the difference? Let's see an example to understand the difference:

File: examples/StaticReturn/LateStaticBinding.php

```
class LateStaticBinding
{
    public static function getSelf()
    {
        return new self();
    }

    public static function getStatic()
    {
        return new static();
    }
}

class LateStaticBindingSubClass extends LateStaticBinding
{

}

echo get_class(LateStaticBindingSubClass::getSelf()) . PHP_EOL;
echo get_class(LateStaticBindingSubClass::getStatic()) . PHP_EOL;
echo get_class(LateStaticBinding::getSelf()) . PHP_EOL;
echo get_class(LateStaticBinding::getStatic()) . PHP_EOL;
```

Code is pretty simple; we defined a class `LateStaticBinding` which return its instance using `self` and `static`. Another class `LateStaticBindingSubClass` extends `LateStaticBinding` class, thus have access to parent class' methods.

Next, we are calling these methods in the instance of two classes. Output is:

```
LateStaticBinding
LateStaticBindingSubClass
LateStaticBinding
LateStaticBinding
```

Got the difference? When we use `self`, it always represents the class it is defined in, `LateStaticBinding` in our example. On the other hand, `static` represents the class from where it is called. Thus, it gives different results, depending on which class' instance it is called.

It is called `late static binding` because it waits until the call and creates/return an instance of the class from where it is called.

Static return type

In later versions (after PHP 5.2), PHP allowed to have type hinting for the return type, but static could not be used there. This will now be possible in PHP 8. Following code demonstrate that, which will work only in PHP 8.

File: examples/StaticReturn/StaticReturnType.php

```
class LateStaticBinding
{
    public static function getSelf(): self
    {
        return new self();
    }

    public static function getStatic(): static
    {
        return new static();
    }
}

class LateStaticBindingSubClass extends LateStaticBinding
{

}

echo get_class(LateStaticBindingSubClass::getSelf()) . PHP_EOL;
echo get_class(LateStaticBindingSubClass::getStatic()) . PHP_EOL;
echo get_class(LateStaticBinding::getSelf()) . PHP_EOL;
echo get_class(LateStaticBinding::getStatic()) . PHP_EOL;
```

Please note, we defined static return type in `public static function getStatic(): static`

Match expression

RFC: https://wiki.php.net/rfc/match_expression_v2

PHP 8 is trying to reduce the line of code and match expression do so in switch cases. Consider this simple switch case example:

File: examples/Match/Switch.php

```
function printFavoriteColor($colour)
{
    switch ($colour) {
        case 'red':
            echo 'Your favorite colour is Red.' . PHP_EOL;
        case 'blue':
            echo 'Your favorite colour is Blue.' . PHP_EOL;
        case 'green':
            echo 'Your favorite colour is Green.' . PHP_EOL;
        default:
            echo 'You do not like any primary colour.' . PHP_EOL;
    }
}

printFavoriteColor('Yellow');
```

A simple program and as you must have correctly expected, the output is `You do not like any primary colour.`

PHP 8 introduce a new `match` expression, which can achieve the same result as above. Let's see that:

File: examples/Match/Match.php

```
function printFavoriteColor($colour)
{
    echo match ($colour) {
        'red' => 'Your favorite colour is Red.' . PHP_EOL,
        'blue' => 'Your favorite colour is Blue.' . PHP_EOL,
        'green' => 'Your favorite colour is Green.' . PHP_EOL,
        default => 'You do not like any primary colour.' . PHP_EOL,
    };
}

printFavoriteColor('Yellow');
```

Shorter code but it will have the same result.

Strict type comparison

Switch loosely compare cases with `==`. Let's see an example.

File: examples/Match/Switch2.php (Example taken from RFC & slightly fixed)

```
switch (false) {
    case 0:
      $result = "Oh no!\n";
      break;
    case false:
      $result = "This is what I expected\n";
      break;
}
echo $result;
```

At a first look, we might be expecting the result to be `This is what I expected` but we get the result as `Oh no!`. Reason is, `0 == false` returns true. Let's see the same example with the match, which does strict comparison `===`.

File: examples/Match/Match2.php

```
echo match (false) {
    0 => "Oh no!\n",
    false => "This is what I expected\n",
};
```

It will return the expected output `This is what I expected`.

No break needed

In switch, we need to `break` each case to stop further execution of switch. Let's check an example:

File: examples/Match/Switch3.php

```php
switch (1) {
    case 1:
    case 2:
        echo 'Number 1 or 2.' . PHP_EOL;
    case 3:
    case 4:
        echo 'Number 3 or 4.' . PHP_EOL;
}
```

What output are we expecting? Isn't it `Number 1 or 2.` but the output is

```
Number 1 or 2.
Number 3 or 4.
```

It's obvious as we forget to `break` before `case 3`. To fix it, we need to add break

File: examples/Match/Switch4.php

```php
switch (1) {
    case 1:
    case 2:
        echo 'Number 1 or 2.' . PHP_EOL;
        break; // Added this line
    case 3:
    case 4:
        echo 'Number 3 or 4.' . PHP_EOL;
}
```

Same example with match do not need break

File: examples/Match/Match3.php

```php
echo match (1) {
    1, 2 => 'Number for 1 and 2' . PHP_EOL,
    3, 4 => 'Number for 3 and 4' . PHP_EOL,
};
```

Return value

In the above example, you may also have noticed that we need to do `echo` in each switch case. However, in case of a match, we echo only once before `match`.

This is due to the fact that switch simply executes the statements but match return the result of statements. This will be especially helpful when we need to return value from `match` in a function. With a switch, we need multiple returns or make a local variable and return the variable after the switch. With the match, we can simply do `return match(){}`.

Mixed types

RFC: https://wiki.php.net/rfc/mixed_type_v2

PHP introduced scalar types in PHP 7, nullable in 7.1, object in 7.2, and lastly, union types in 8.0 as discussed above. Many developers started using these features and now type-hinting their parameters and return types.

However, we may still see this type-hinting is missing due to different reasons like:

- the type is a specific type, but the `programmer forgot` to declare it.
- the type is a specific type, but the programmer omitted it to keep `compatibility with an older PHP version`
- the `type is not currently expressible` in PHP's type system, and so no type could be specified.
- for return types, it is `not clear if the function will or will not return a value, other than null`.

Reason for mixed types: A mixed type would allow people to add types to parameters, class properties and the function returns to indicate that the type information wasn't forgotten about, it just can't be specified more precisely, or the programmer explicitly decided not to do so.

Because of the reasons above, it's a good thing the mixed type is added. Mixed itself means one of these types:

- array
- bool
- callable
- int
- float
- null
- object

- resource
- string

Example

```
function someMethod(): mixed
```

Also note that since mixed already includes null, it's not allowed to make it nullable. The following will trigger an error:

```
function someMethod(): ?mixed
```

Throw expression

RFC: https://wiki.php.net/rfc/throw_expression

Throw in PHP 7 and earlier version was a statement.

Statement vs expression

- Stagement: A line of code which does something like `for`, `if`, etc.
- Expression: A line of code which evalute something like `1 + 2`, `$a == $b` (evalute true of false), etc.

Throw statement

In PHP version 7 and earlier, the throw was a statement, that means, it cannot be used at places where expression is needed. Examples (Taken from RFC)

```
// This was previously not possible since arrow functions only accept a
single expression while throw was a statement.
$callable = fn() => throw new Exception();

// $value is non-nullable.
$value = $nullableValue ?? throw new InvalidArgumentException();

// $value is truthy.
$value = $falsableValue ?: throw new InvalidArgumentException();

// $value is only set if the array is not empty.
$value = !empty($array)
    ? reset($array)
    : throw new InvalidArgumentException();
```

Above lines of code will generate errors in PHP 7 and earlier.

PHP 8 made `throw statement` to `throw expression` and above code will work.

Operator precedence

Now since throw is an expression (like operators), its position in operator precedence is important. Below examples (Taken from RFC) will explain operator precedence of throw expression.

```
throw $this->createNotFoundException();
// Evaluated as
throw ($this->createNotFoundException());
// Instead of
(throw $this)->createNotFoundException();

throw static::createNotFoundException();
// Evaluated as
throw (static::createNotFoundException());
// Instead of
(throw static)::createNotFoundException();

throw $userIsAuthorized ? new ForbiddenException() : new
UnauthorizedException();
// Evaluated as
throw ($userIsAuthorized ? new ForbiddenException() : new
UnauthorizedException());
```

```
// Instead of
(throw $userIsAuthorized) ? new ForbiddenException() : new
UnauthorizedException();

throw $maybeNullException ?? new Exception();
// Evaluated as
throw ($maybeNullException ?? new Exception());
// Instead of
(throw $maybeNullException) ?? new Exception();

throw $exception = new Exception();
// Evaluated as
throw ($exception = new Exception());
// Instead of
(throw $exception) = new Exception();

throw $exception ??= new Exception();
// Evaluated as
throw ($exception ??= new Exception());
// Instead of
(throw $exception) ??= new Exception();

throw $condition1 && $condition2 ? new Exception1() : new Exception2();
// Evaluated as
throw ($condition1 && $condition2 ? new Exception1() : new
Exception2());
// Instead of
(throw $condition1) && $condition2 ? new Exception1() : new
Exception2();
```

Important: Keeping throw at lower precedence will have a side effect, `throw between two short-circuit operators would not be possible without parentheses`. Example:

```
$condition || throw new Exception('$condition must be truthy')
  && $condition2 || throw new Exception('$condition2 must be truthy');
// Evaluated as
$condition || (throw new Exception('$condition must be truthy') &&
$condition2 || (throw new Exception('$condition2 must be truthy')));
// Instead of
$condition || (throw new Exception('$condition must be truthy'))
  && $condition2 || (throw new Exception('$condition2 must be truthy'));
```

Although it will be a rare case, if we decide to use this, we must use proper parentheses.

Stringable interface

RFC: https://wiki.php.net/rfc/stringable

PHP 8 introduced a new interface `Stringable`.

```
interface Stringable
{
   public function __toString(): string;
}
```

Usage example

```
class ClassWithToString
{
    public function __toString(): string
    {
        return 'test string from ClassWithToString::__toString';
    }
}

function testStringable(string|Stringable $stringable) {
    echo "String is '$stringable'" . PHP_EOL;
}

testStringable(new ClassWithToString());
testStringable('Notmal string');
```

Output

```
String is 'test string from ClassWithToString::__toString'
String is 'Normal string'
```

Please note, we have not used `implements Stringable` in our class `ClassWithToString`. PHP 8 automatically add `implements Stringable` to any class that have `__toString()` method.

Validation for abstract trait methods

RFC: https://wiki.php.net/rfc/abstract_trait_method_validation

PHP 8 will strictly validate the signature of a trait's abstract method. Following code was possible in PHP 7:

File: None. (Taken from RFC)

```
trait T {
    abstract public function test(int $x);
}

class C {
    use T;

    // Allowed in PHP 7 but not in PHP 8.
    public function test(string $x) {}
}
```

Now in PHP 8, above code is not valid because parameter type (int) of class C does not match with parameter type (string) of trait T. To make it work in PHP 8, parameter $x in C::test must be of type `int`. The following example will run in PHP 8

File: None. (Improved RFC example)

```
trait T {
    abstract public function test(int $x);
}

class C {
    use T;

    // This will work in PHP 8.
    public function test(int $x) {}
}
```

New functions

PHP 8 introduced lot of new internal functions. Let's see them one by one.

Function `str_contains`

PHP 8 added a new function `str_contains`, which checks if a string is contained in another string and returns a boolean value (true/false) whether or not the string was found.

Syntax

```
str_contains ( string $haystack , string $needle ) : bool
```

Example

```
str_contains("abc", "a"); // true
str_contains("abc", "d"); // false

// $needle is an empty string
str_contains("abc", "");  // true
str_contains("", "");     // true
```

Note from RFC: As of PHP 8, the behaviour of "" in string search functions is well defined, and we consider "" to occur at every position in the string, including one past the end. As such, both of these will (or at least should) return true. The empty string is contained in every string.

Function `str_starts_with` and `str_ends_with`

RFC: https://wiki.php.net/rfc/add_str_starts_with_and_ends_with_functions

str_starts_with checks if a string begins with another string and returns a boolean value (true/false) whether it does.

str_ends_with checks if a string ends with another string and returns a boolean value (true/false) whether it does.

Syntax

```
str_starts_with ( string $haystack , string $needle ) : bool
str_ends_with ( string $haystack , string $needle ) : bool
```

Example (Taken from RFC)

```
$str = "beginningMiddleEnd";
if (str_starts_with($str, "beg")) echo "printed\n";
if (str_starts_with($str, "Beg")) echo "not printed\n";
if (str_ends_with($str, "End")) echo "printed\n";
if (str_ends_with($str, "end")) echo "not printed\n";

// empty strings:
if (str_starts_with("a", "")) echo "printed\n";
if (str_starts_with("", "")) echo "printed\n";
if (str_starts_with("", "a")) echo "not printed\n";
if (str_ends_with("a", "")) echo "printed\n";
if (str_ends_with("", "")) echo "printed\n";
if (str_ends_with("", "a")) echo "not printed\n";
```

Function `get_debug_type`

RFC: https://wiki.php.net/rfc/get_debug_type

PHP already have `[gettype](https://www.php.net/manual/en/function.gettype.php)` since version 4. It returns the type of a variable for a given variable.

PHP 8 introduced new function `get_debug_type`, it will be same as `gettype` + it will also work on objects. Example:

File: examples/NewFunctions/get_debug_type.php

```php
class ExampleClass { }

$exampleClassOrBool1 = new ExampleClass();
$exampleClassOrBool2 = false;

echo "PHP 7 example" . PHP_EOL;

echo "exampleClassOrBool1 is ";
echo is_object($exampleClassOrBool1)
        ? get_class($exampleClassOrBool1)
        : gettype($exampleClassOrBool1);
echo PHP_EOL;

echo "exampleClassOrBool2 is ";
echo is_object($exampleClassOrBool2)
        ? get_class($exampleClassOrBool2)
        : gettype($exampleClassOrBool2);
echo PHP_EOL;

echo "PHP 8 example" . PHP_EOL;

echo "exampleClassOrBool1 is " . get_debug_type($exampleClassOrBool1) .
PHP_EOL;
echo "exampleClassOrBool2 is " . get_debug_type($exampleClassOrBool2) .
PHP_EOL;
```

Possible output for `get_debug_type` and `gettype` for different inputs can be seen at RFC.

Function `fdiv`

RFC: Not able to find. PR: https://github.com/php/php-src/pull/4769

PHP support `fmod` since PHP 4 and `intdiv` since version 7.

In the same family of functions, PHP 8 added `fdiv`. It is similar to `intdiv` but works for float.

Syntax changes

Allowing `::class` on objects

RFC: https://wiki.php.net/rfc/class_name_literal_on_object

With PHP 8, we can now use `::class` on objects.

Till PHP 7, `::class` was possible only on a class name. To get a class of an object, we need to call a function `get_class($object)`. Now with PHP 8, we can do `$object::class`.

Non-capturing catches

PHP, till version 7, requires to capture the exception being caught to a variable.

```
try {
    foo();
} catch (SomeException $ex) {
    die($ex->getMessage());
}
```

In the above case, we are using `$ex`. However, even if we do not need to use an exception object, we still needed to define it in PHP 7 and earlier versions like

```
try {
    changeImportantData();
} catch (PermissionException $ex) {
    echo "You don't have permission to do this";
}
```

With PHP 8, if we do not want to use a variable, we need not assign it to a variable. Example

```
try {
    changeImportantData();
} catch (PermissionException) { // The intention is clear: exception 
details are irrelevant
    echo "You don't have permission to do this";
}
```

Allow a trailing comma in the parameter list

RFC: https://wiki.php.net/rfc/trailing_comma_in_parameter_list

PHP already supported adding a comma after the last element of an array like

```
$exampleArray = [
    'a',
    'b',
    'c', // Comma after lase element is allowed in PHP
];
```

Now with PHP 8, it is also possible with the parameter list of a method.

```
new Uri(
    $scheme,
    $user,
    $pass,
    $host,
    $port,
    $path,
    $query,
    $fragment, // <-- Huh, this is allowed!
);
```

Breaking changes

Link any major version, PHP 8 also introduced some breaking changes. Thus, we need to look at them and fix our existing projects, before upgrading PHP to version 8 on the server.

Once PHP 8 is released, documentation will cover PHP 7 to PHP 8 migration guide, with breaking changes. However, until documentation is available in PHP manual, we can see it at https://github.com/php/php-src/blob/PHP-8.0/UPGRADING#L20

Consistent type errors for internal functions

RFC: https://wiki.php.net/rfc/consistent_type_errors

For user-defined functions, passing a parameter of illegal type results in a TypeError. For internal functions, the behaviour depends on multiple factors, but the default is to throw a warning and return null.

Example of user-defined function in PHP 7

```
function foo(int $bar) {}
foo("not an int");
// TypeError: Argument 1 passed to foo() must be of the type int, string
given
```

Example of internal function in PHP 7

```
var_dump(strlen(new stdClass));
// Warning: strlen() expects parameter 1 to be string, object given
// NULL
```

However we can change this behaviour by using strict_types in PHP 7.

```
declare(strict_types=1);
var_dump(strlen(new stdClass));
// TypeError: strlen() expects parameter 1 to be string, object given
```

In PHP 8, internal parameter parsing APIs always generate a TypeError if parameter parsing fails.

Change Default PDO Error Mode

RFC: https://wiki.php.net/rfc/pdo_default_errmode

The current default error mode for PDO is silent. This means that when an SQL error occurs, no errors or warnings may be emitted and no exceptions thrown unless the developer implements their own explicit error handling.

This causes issues for new developers because the only errors they often see from PDO code are knock-on errors such as “call to fetch() on non-object” - there's no indication that the SQL query (or other action) failed or why.

Change the precedence of the concatenation operator

RFC: https://wiki.php.net/rfc/concatenation_precedence

It's been a long standing issue that an (unparenthesized) expression with '+', '-' and '.' evaluates left-to-right.

```
echo "sum: " . $a + $b;

// current behavior: evaluated left-to-right
echo ("sum: " . $a) + $b;

// desired behavior: addition and subtraction have a higher precendence
echo "sum :" . ($a + $b);
```

Stricter type checks for arithmetic/bitwise operators

RFC: https://wiki.php.net/rfc/arithmetic_operator_type_checks

This RFC proposes to throw a TypeError when arithmetic or bitwise operator is applied to an array, resource or (non-overloaded) object. The behaviour of scalar operands (like null) remains unchanged

This is not reasonable behaviour:

```
var_dump([] % [42]);
// int(0)
```

Saner numeric strings

RFC: https://wiki.php.net/rfc/saner-numeric-strings

The PHP language has a concept of numeric strings, strings which can be interpreted as numbers.

A string can be categorised in three ways

- A numeric string is a string containing only a number, optionally preceded by whitespace characters. For example, "123" or " 1.23e2".
- A leading-numeric string is a string that begins with a numeric string but is followed by non-number characters (including whitespace characters). For example, "123abc" or "123 ".
- A non-numeric string is a string which is neither a numeric string nor a leading-numeric string.

PHP 8 will unify the various numeric string modes into a single concept: Numeric characters only with both leading and trailing whitespace allowed. Any other type of string is non-numeric and will throw TypeErrors when used in a numeric context.

Saner string to number comparisons

RFC: https://wiki.php.net/rfc/string_to_number_comparison

Comparisons between strings and numbers using == and other non-strict comparison operators currently work by casting the string to a number, and subsequently performing a comparison on integers or floats. This results in many surprising comparison results, the most notable of which is that 0 == "foobar" returns true.

The single largest source of bugs is likely the fact that `0 == "foobar"` returns true. Quite often this is encountered in cases where the comparison is implicit, such as in_array() or switch statements. A classic example:

```
$validValues = ["foo", "bar", "baz"];
$value = 0;
var_dump(in_array($value, $validValues)); // bool(true) WTF???
```

PHP 8 will give the string to number comparisons a more reasonable behaviour: When comparing to a numeric string, use a number comparison (same as now). Otherwise, convert the number to string and use a string comparison. Examples:

```
var_dump(0 == "0");      // PHP 7: true; PHP 8: true
var_dump(0 == "0.0");    // PHP 7: true; PHP 8: true
var_dump(0 == "foo");    // PHP 7: true; PHP 8: false
var_dump(0 == "");       // PHP 7: true; PHP 8: false
var_dump(42 == "   42"); // PHP 7: true; PHP 8: true
var_dump(42 == "42foo"); // PHP 7: true; PHP 8: false
```

Make sorting stable

RFC: https://wiki.php.net/rfc/stable_sorting

Sorting functions in PHP are currently unstable, which means that the order of “equal” elements is not guaranteed. Example:

```
$array = [
    'c' => 1,
    'd' => 1,
    'a' => 0,
    'b' => 0,
];
asort($array);
```

With stable sorting, we might expect output to be:

```
['a' => 0, 'b' => 0, 'c' => 1, 'd' => 1]
```

However, since PHP 7 sorting is currently unstable, it could be one of the following as well

```
['b' => 0, 'a' => 0, 'c' => 1, 'd' => 1]
['a' => 0, 'b' => 0, 'd' => 1, 'c' => 1]
['b' => 0, 'a' => 0, 'd' => 1, 'c' => 1]
```

PHP 8 will have stable sorting for sorting functions like: sort, rsort, usort, asort, arsort, uasort, ksort, krsort, uksort, array_multisort.

Reclassifying engine warnings

RFC: https://wiki.php.net/rfc/engine_warnings

A lot of warning in PHP 7 will now be converted to Errors (Mostly Error exception or Type Error)

Please check the RFC to see changes in all warnings of PHP 7 which are converted to Errors in PHP 8.

Internal Improvements

JIT

Inheritance with private methods

`ext-json` always available