What's New in PHP 8.1/8.2?

PHP UK Conference

Derick Rethans

derick@php.net — @derickr@phpc.social

https://derickrethans.nl/talks/php-phpuk23

About Me

Derick Rethans

- I'm European, living in London
- ~~PHP 7.4 **Release Manager**~~
- **PHP Foundation**
- **Xdebug** & PHP's Date/Time support
- I ♥🌍 maps, I ♥🍺 beer, I ♥🥃 whisky
- mastodon: @derickr@phpc.social
- twitter: @derickr

Union Types

```php
<?php
declare(strict_types=1);

class Number {



    private $number;



    public function setNumber($number) {
        $this->number = $number;
    }



    public function getNumber() {
        return $this->number;
    }
}
```

Union Types

```php
<?php
declare(strict_types=1);

class Number {
    /**
     * @var int|float $number
     */
    private $number;

    /**
     * @param int|float $number
     */
    public function setNumber($number) {
        $this->number = $number;
    }

    /**
     * @return int|float
     */
    public function getNumber() {
        return $this->number;
    }
}
```

Union Types

```php
<?php
declare(strict_types=1);

class Number {



    private int|float $number;




    public function setNumber(int|float $number) {
        $this->number = $number;
    }



    public function getNumber() : int|float {
        return $this->number;
    }
}
```

Union Types — Examples

```
function microtime(bool $get_as_float = false): string|float {}

function gettimeofday(bool $return_float = false): array|float {}

function base64_decode(string $str, bool $strict = false): string|false {}

function implode(string|array $glue, array $pieces): string {}

function substr_replace(
    string|array $str,
    string|array $replace,
    $start,
    $length
) : string|array|false {}
```

Union Types — Rules

All support types can be part of a union, with the following caveats:

- **`void`** can't be part of a union type
- **`?Class`** is equivalent to **`Class|null`**
- **`false`** is an addition, as it is used in the PHP standard library
  e.g.: **`strpos() : int|false`**
- No redundant types, such as: **`bool|false`**, or: **`object|Whisky`**

Variance:

- Union types follow standard variance rules
- Order is irrelevant: **`int|string`** and **`string|int`**
- **`array|Traversable`** is the same as **`iterable`**

Readonly Properties

To enforce 'readonly'-bility:

```
<?php
class User
{
    public function __construct(private string $name) {}

    public function getName(): string {
        return $this->name;
    }
}
```

Readonly Properties

In PHP 8.1, with a real keyword, `readonly`:

```
<?php
class User
{
    public function __construct(public readonly string $name) {}
}
```

Readonly Properties

In PHP 8.1, with a real keyword, `readonly`:

```php
<?php
class User
{
    public function __construct(public readonly string $name) {}
}
```

Restrictions:

- Can only be initialised once
- Can only be initialised from the scope where it was declared

Readonly Classes

From PHP 8.2, marking all properties as `readonly`:

```php
<?php
readonly class User
{
    public function __construct(public string $name) {}
}
```

Additionally:

- Prevents dynamic properties from being created
- Can't be used if there are untyped or static properties

`mixed`

`array|bool|callable|int|float|null|object|resource|string`

- Distinguishes between:
  - I didn't add the type yet
  - It really accepts any value
- Acts as a union type
- Follows standard variance rules
- A missing return type declaration is assumed to be: **`mixed|void`**

Constructor Property Promotion

```php
<?php
class 🥃whisky
{
    public    string      $name;
    protected Distillery  $distillery;
    public    int         $age;
    private   ?Bottler    $bottler = null;
```

```php
    function __construct(string $name, Distillery $distillery, int $age, ?Bottler $
    {
        $this->name = $name;
        $this->distillery = $distillery;
        $this->age = $age;
        $this->bottler = $bottler;
    }
```

Constructor Property Promotion

```php
<?php
class 🥃whisky
{
    function __construct(
        public    string $name,
        protected Distillery $distillery,
        public    int $age,
        private   ?Bottler $bottler = null,
    ) {}
}
```

Object Instantiation

```
<?php
class Settings
{
    function __construct(
        public bool   $includeDistillery   = true,
        public bool   $includeDescription  = true,
        public bool   $includeTastingNotes = false,
        public int    $pageSize            = 64,
        public string $sortByField         = 'whisky',
    ) {}
}
```

Instantiation with different includes:

```
$settings = new \Settings(true, false);
```

Object Instantiation

```
<?php
class Settings
{
    function __construct(
        public bool   $includeDistillery   = true,
        public bool   $includeDescription  = true,
        public bool   $includeTastingNotes = false,
        public int    $pageSize            = 64,
        public string $sortByField         = 'whisky',
    ) {}
}
```

Instantiation with different includes:

```
$settings = new \Settings(true, false);
```

Instantiation with different sort order:

```
$settings = new \Settings(true, true, false, 64, 'rating');
```

Object Instantiation with Named Arguments

```
<?php
class Settings
{
    function __construct(
        public bool   $includeDistillery   = true,
        public bool   $includeDescription  = true,
        public bool   $includeTastingNotes = false,
        public int    $pageSize            = 64,
        public string $sortByField         = 'whisky',
    ) {}
}
```

Instantiation with different includes:

```
$settings = new \Settings(includeDescription: false);
```

Object Instantiation with Named Arguments

```
<?php
class Settings
{
    function __construct(
        public bool   $includeDistillery   = true,
        public bool   $includeDescription  = true,
        public bool   $includeTastingNotes = false,
        public int    $pageSize            = 64,
        public string $sortByField         = 'whisky',
    ) {}
}
```

Instantiation with different includes:

```
$settings = new \Settings(includeDescription: false);
```

Instantiation with different sort order:

```
$settings = new \Settings(sortByField: 'rating');
```

Enumerations

`enum`

a data type consisting of a set of named values

Enumerations

```php
<?php
enum Currency {
    case GBP;
    case EUR;
}
```

Enumerations

```php
<?php
enum Currency {
    case GBP;
    case EUR;
}
```

```php
class Product {
    function __construct(
        private string $name, private float $amount, private Currency $currency) {
}

$elephpant = new Product( 'Chilli', 5.50, Currency::EUR );
var_dump( $elephpant );
```

Enumerations

```php
<?php
enum Currency {
    case GBP;
    case EUR;
}
```

```php
class Product {
    function __construct(
        private string $name, private float $amount, private Currency $currency) {
}

$elephpant = new Product( 'Chilli', 5.50, Currency::EUR );
var_dump( $elephpant );
```

```
class Product#1 (3) {
  private string $name => string(6) "Chilli"
  private float $amount => double(5.5)
  private Currency $currency => enum Currency::EUR;
}
```

Enumerations — Typed

```php
<?php
enum Currency {
    case GBP;
    case EUR;
}

class Product {
    function __construct(
        private string $name, private float $amount, private Currency $currency) {
}

$elephpant = new Product( 'Chilli', 5.50, "EUR" );
```

```
TypeError: Product::__construct(): Argument #3 ($currency) must be of type Currency
    string given, called in /tmp/enum-2.php on line 12 in /tmp/enum-2.php on line 8
```

Enumerations — Backed

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}

class Product {
    function __construct(
        private string $name, private float $amount, private Currency $currency) {
}

$elephpant = new Product( 'Chilli', 5.50, Currency::GBP );
var_dump( $elephpant );
```

```
class Product#1 (3) {
  private string $name => string(6) "Chilli"
  private float $amount => double(5.5)
  private Currency $currency => enum Currency::GBP : string("£");
}
```

Enumerations — Backed Values

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}
```

Value Property:

```php
var_dump( Currency::EUR->value );
```

```
string(3) "€"
```

Enumerations — Backed Values

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}
```

Create from Value:

```php
var_dump( Currency::from('£') );
```

```
enum Currency::GBP : string("£");
```

Enumerations — Backed Values

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}
```

Try Create from Value:

```php
var_dump( Currency::tryFrom('$') );
```

```
NULL
```

Enumerations — Functions

```
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';

    public function format( float $amount )
    {
        $pattern = match( $this ) {
            Currency::GBP => "{$this->value} %.2f",
            Currency::EUR => "%.2f {$this->value}",
        };

        return sprintf( $pattern, $amount );
    }
}
```

Calling Methods:

```
$cur = Currency::GBP;
echo $cur->format( 42.75 );
```

```
£ 42.75
```

Enumerations — Static Methods

```
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';

    static public function fromLocale( string $locale )
    {
        return match( $locale ) {
            'nl_NL' => static::EUR,
            'it_IT' => static::EUR,
            'uk_GB' => static::GBP,
        };
    }
}
```

Alternative Constructor:

```
$cur = Currency::fromLocale( 'it_IT' );
echo $cur->value, "\n";
```

```
€
```

Enumerations — Value Listing

```
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}
```

List Cases:

```
var_dump( Currency::cases() );
```

```
array(2) {
  [0]=>
  enum(Currency::GBP)
  [1]=>
  enum(Currency::EUR)
}
```

Enumerations — Fetch Properties

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}

const PRICES = [
        Currency::EUR => 19.99,
        Currency::GBP => 17.99,
];
```

PHP 8.1:

```
Fatal error: Constant expression contains invalid operations
```

PHP 8.2:

```
Fatal error: Constant expression contains invalid operations
```

Enumerations — Fetch Properties

```php
<?php
enum Currency: string {
    case GBP = '£';
    case EUR = '€';
}

const PRICES = [
        Currency::EUR->value => 19.99,
        Currency::GBP->value => 17.99,
];
```

The 'never' Type

Explicitly mark a function that it never returns:

```
<?php
function redirect(string $uri): never {
    header('Location: ' . $uri);
    exit();
}
```

Or implicitily:

```
function redirectToLoginPage(): never {
    redirect('/login');
}
```

The 'never' Type

Explicitly mark a function that it never returns:

```
<?php
function redirect(string $uri): never {
    header('Location: ' . $uri);
    exit();
}
```

Or implicitily:

```
function redirectToLoginPage(): never {
    redirect('/login');
}
```

Remarks:

- Just like `void`, only as a return type
- It is a **bottom** type

Pure Intersection Types

To enforce a value is both `Traversable` and `Countable`:

```
<?php
class Test {
    private ?Traversable $traversable = null;
    private ?Countable $countable = null;
    /** @var Traversable&Countable */
    private $both = null;

    public function __construct($countableIterator) {
        $this->traversable =& $this->both;
        $this->countable =& $this->both;
        $this->both = $countableIterator;
    }
}
```

Pure Intersection Types

In PHP 8.1:

```php
<?php
class Test {
    private Traversable&Countable $countableIterator;

    public function setIterator(Traversable&Countable $countableIterator): void {
        $this->countableIterator = $countableIterator;
    }

    public function getIterator(): Traversable&Countable {
        return $this->countableIterator;
    }
}
```

Pure Intersection Types

In PHP 8.1:

```php
<?php
class Test {
    private Traversable&Countable $countableIterator;

    public function setIterator(Traversable&Countable $countableIterator): void {
        $this->countableIterator = $countableIterator;
    }

    public function getIterator(): Traversable&Countable {
        return $this->countableIterator;
    }
}
```

- Only for class and interface names
- Variance rules are respected, but they are complicated

Array Unpacking with String Keys

Added in PHP 7.4:

```
<?php
$apples = [ '🍎',  '🍏' ];
$fruits = [ '🍐', '🍑', ...$apples, '🍓',  '🍅' ];

echo "Fruit salad: ", implode( '', $fruits ), "\n";
?>
```

Result:

```
Fruit salad: 🍐🍑🍎🍏🍓🍅
```

Array Unpacking with String Keys

PHP 8.1 adds support for string keys

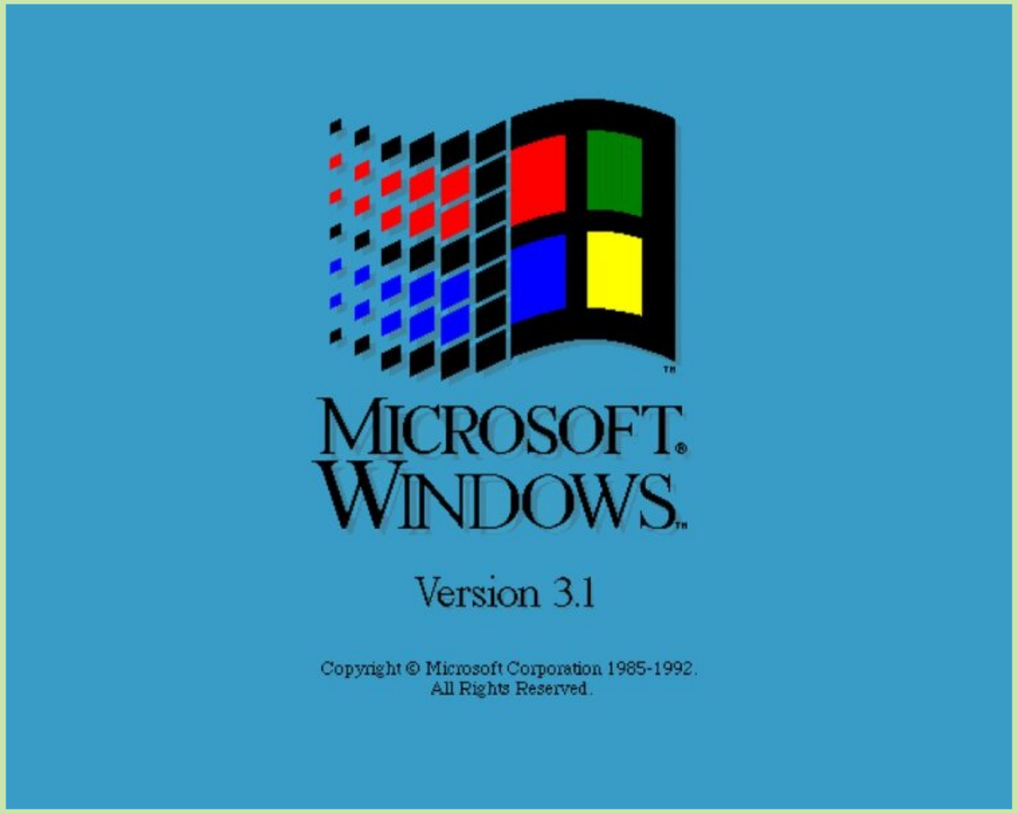
```
<?php
$apples = [ 'red' => '🍎',  'green' => '🍏' ];
$others = [ 'green' => '🍐', 'pink' => '🍑' ];
$fruits = [ ...$apples, ...$others ];

var_dump( $fruits );
?>
```

Result:

```
array(3) {
  'red' => string(4) "🍎"
  'green' => string(4) "🍐"
  'pink' => string(4) "🍑"
}
```

Co-operative Multi-Tasking



First Class Callable Syntax

```php
<?php
class Test {
    public function getPrivateMethod() {
        return [$this, 'privateMethod'];
        return Closure::fromCallable([$this, 'privateMethod']);
        return $this->privateMethod(...);
    }

    private function privateMethod() {
        echo __METHOD__, "\n";
    }
}

$test = new Test;
$privateMethod = $test->getPrivateMethod();
$privateMethod();
?>
```

First Class Callable Syntax

```php
<?php
class Test {
    public function getPrivateMethod() {
        return [$this, 'privateMethod'];
        return Closure::fromCallable([$this, 'privateMethod']);
        return $this->privateMethod(...);
    }

    private function privateMethod() {
        echo __METHOD__, "\n";
    }
}

$test = new Test;
$privateMethod = $test->getPrivateMethod();
$privateMethod();
?>
```

First Class Callable Syntax

```php
<?php
class Test {
    public function getPrivateMethod() {
        return [$this, 'privateMethod'];*
        return Closure::fromCallable([$this, 'privateMethod']);
        return $this->privateMethod(...);
    }

    private function privateMethod() {
        echo __METHOD__, "\n";
    }
}

$test = new Test;
$privateMethod = $test->getPrivateMethod();
$privateMethod();
?>
```

First Class Callable Syntax

Syntactic Sugar

```
$fn = Closure::fromCallable('strlen');
$fn = strlen(...);

$fn = Closure::fromCallable([$this, 'method']);
$fn = $this->method(...)

$fn = Closure::fromCallable([Foo::class, 'method']);
$fn = Foo::method(...);
```

Other variants

```
strlen(...);
$closure(...);
$obj->method(...);
$obj->$methodStr(...);
($obj->property)(...);
Foo::method(...);
$classStr::$methodStr(...);
self::{$complex . $expression}(...);
[$obj, 'method'](...);
[Foo::class, 'method'](...);
```

PHP 8.2: `null` and `false` types

- `false` could only be used as part of a union (i.e. `string|false`)
- But not as `null|false` and had to use `null|bool`
- You could not use `null` standalone, even though it's a narrower type:

```
<?php
class User {}

class UserFinder {
    function findUserByEmail(string $email) : User|null {}
}

class AlwaysNullUserFinder extends UserFinder {
    function findUserByEmail(string $email) : ~~null~~ {}
}
?>
```

`void` means: Does not return **anything**

`null` means: Does return the **null** value

PHP 8.2: `true` type

We had `false` and `null`, but not `true`

PHP 8.2: DNF Types

Union Types: **X|Y**

```
class Number {
    private int|float $number;

    public function setNumber(int|float $number): void {
        $this->number = $number;
    }

    public function getNumber(): int|float {
        return $this->number;
    }
}
```

(Pure) Intersection Types: **X&Y**

Disjunctive Normal Form Types: **(X&A)|(Y&B)**

PHP 8.2: DNF Types

Union Types: **X|Y**

(Pure) Intersection Types: **X&Y**

```
class Iteraty {
    private Traversable&Countable $countableIterator;

    public function setIterator(Traversable&Countable $countableIterator): void {
        $this->countableIterator = $countableIterator;
    }

    public function getIterator(): Traversable&Countable {
        return $this->countableIterator;
    }
}
```

Disjunctive Normal Form Types: **(X&A)|(Y&B)**

PHP 8.2: DNF Types

Union Types: **X|Y**

(Pure) Intersection Types: **X&Y**

Disjunctive Normal Form Types: **(X&A)|(Y&B)**

```
class Foreachy {
    private array|(Traversable&Countable) $foreachableData;

    public function setIterator(array|(Traversable&Countable) $foreachable): void
        $this->foreachableData = $foreachable;
    }

    public function getIterator(): array|(Traversable&Countable) {
        return $this->foreachableData;
    }
}
```

PHP 8.2: DNF Types

Union Types: **X|Y**

(Pure) Intersection Types: **X&Y**

Disjunctive Normal Form Types: **(X&A)|(Y&B)**

New Random extension

```php
<?php
$engine = new \Random\Engine\Xoshiro256StarStar( /* 42 */ );
$r = new \Random\Randomizer($engine);

echo bin2hex( $r->getBytes( 8 ) ), "\n";

echo $r->getInt(1, 100), "\n";
echo $r->nextInt(), "\n";

$fruits = ['red' => '🍎', 'green' => '🥝', 'yellow' => '🍌', 'purple' => '🍇'];
echo "Keys: ", join( ', ', $r->pickArrayKeys( $fruits, 2 ) ), "\n";

echo "Salad: ", join( ', ', $r->shuffleArray( $fruits ) ), "\n";

echo $r->shuffleBytes( "PHP is great!" ), "\n";
?>
```

Result:

```
cb892b47ae582131
8
5341693838347761702
Keys: green, purple
Salad: 🍎, 🍌, 🥝, 🍇
set!HPaiPg r
```

Deprecating Dynamic Properties

```php
<?php
class ezcGraphDataSet
{
    protected $pallet;

    function setPalette( $palette )
    {
        $this->palette = $palette;
    }
}

$e = new ezcGraphDataSet;
$e->setPalette( 'rgb' );
```

```
Deprecated: Creation of dynamic property ezcGraphDataSet::$palette is deprecated
```

Deprecating Dynamic Properties

```php
<?php
class ezcGraphDataSet
{
    protected $pallet;

    function setPalette( $palette )
    {
        $this->palette = $palette;
    }
}

$e = new ezcGraphDataSet;
$e->setPalette( 'rgb' );
```

```
Deprecated: Creation of dynamic property ezcGraphDataSet::$palette is deprecated
```

```
--- src/datasets/base.php
+++ src/datasets/base.php
@@ -83,7 +83,7 @@ abstract class ezcGraphDataSet implements ArrayAccess, Iterator, Countable
      *
      * @var ezcGraphPalette
      */
-    protected $pallet;
+    protected $palette;

    /**
     * Array keys
```

Sensitive Parameters

```php
<?php
function logIn($userName, #[\SensitiveParameter] $password) {
    throw new \Exception('Error');
}

logIn( 'derick', 'secret-elephpant' );
```

Without Xdebug:

```
Fatal error: Uncaught Exception: Error in Standard input code:4
Stack trace:
#0 Standard input code(7): logIn('derick', Object(SensitiveParameterValue))
#1 {main}
  thrown in Standard input code on line 4
```

Sensitive Parameters

```php
<?php
function logIn($userName, #[\SensitiveParameter] $password) {
    throw new \Exception('Error');
}

logIn( 'derick', 'secret-elephpant' );
```

With Xdebug:

```
Fatal error: Uncaught Exception: Error in Standard input code on line 4

Exception: Error in Standard input code on line 4

Call Stack:
  4.8626  399576  1. {main}() Standard input code:0
  4.8626  399576  2. logIn($userName = 'derick', $password = '[Sensitive Parameter
```

Any Queries?

Resources

Slides

`https://derickrethans.nl/talks/php-phpuk23`

`https://xdebug.cloud`

Derick Rethans — @derickr@phpc.social — derick@php.net