

The background is a vibrant, stylized illustration of an underwater scene. It features a blue central area representing water, surrounded by green and orange sections. In the top left, there is a large, orange, branching coral-like shape. To its right, a school of yellow fish swims towards the right. Below the fish, there are several red circles of varying sizes, resembling bubbles. In the bottom left corner, there is a green, branching coral-like shape. In the bottom right corner, there are three red, curved lines that look like seaweed or a fish's tail. The overall style is flat and modern.

# PHP 8 in a Nutshell

Amit D. Merchant

[amitmerchant.com](http://amitmerchant.com)

# Table of Contents

<b>Introduction</b>	<b>11</b>
<b>PHP 8</b>	<b>12</b>
<b>Union Types</b>	<b>13</b>
What are union types exactly?	13
Advantages of using union types	15
Scope of union types	15
<b>The Nullsafe Operator</b>	<b>16</b>
The nullsafe operator	16
<b>Constructor Property Promotion</b>	<b>18</b>
What is Constructor Property Promotion?	18
Rules of using constructor property promotion	19
<b>New String functions</b>	<b>21</b>
The str_contains function	21
str_starts_with and str_ends_with	23
<b>Non-capturing Exception Catches</b>	<b>25</b>
Introducing non-capturing catches	25
<b>The Mixed Type</b>	<b>27</b>
What is a Mixed type?	27

<b>Match Expressions</b>	<b>34</b>
Good ol' switch statement	34
Match expression	35
<b>Named Arguments</b>	<b>39</b>
Named parameters	39
In closing	41
<b>Attributes</b>	<b>42</b>
What are Attributes?	42
How to define Attributes?	43
Practical usage	44
Scopes of Attributes	46
Real-world usage	47
<b>The ::class keyword</b>	<b>49</b>
Using ::class on objects in PHP 8	49
<b>Throw as an Expression</b>	<b>51</b>
The throw keyword as an expression	51
<b>The get_debug_type() function</b>	<b>54</b>
The get_debug_type() function	55
In closing	56
<b>PHP 8.1</b>	<b>57</b>
<b>Readonly Properties</b>	<b>58</b>
The old way	58

Readonly properties in PHP 8.1 .....	60
In closing .....	62
<b>Native Enumerations .....</b>	<b>64</b>
Enums in PHP 8.1 .....	64
Fetch Enum Cases .....	67
Advantage of using enums .....	67
In Closing .....	68
<b>Fibers or Coroutines .....</b>	<b>69</b>
Fibers or Coroutines or Green threads in PHP 8.1 .....	69
Creating a Fiber .....	69
In closing .....	71
<b>Intersection Types .....</b>	<b>72</b>
Pure Intersection Types .....	72
Some Gotchas .....	73
<b>First-class callables .....</b>	<b>74</b>
What are First-class callables? .....	74
A practical example .....	74
<b>New in initializers .....</b>	<b>77</b>
Initializing objects in constructor properties .....	78
Restrictions .....	79
In closing .....	80
<b>Array unpacking with string keys .....</b>	<b>81</b>
The old way .....	81
Array unpacking with string keys .....	82

<b>The array_is_list() function .....</b>	<b>83</b>
The array_is_list() function .....	84
The Caveat .....	85
 <b>The Never Return Type .....</b>	 <b>86</b>
The never return type .....	86
Gotchas .....	87
 <b>Final class constants .....</b>	 <b>89</b>
The final class constants .....	90
 <b>PHP 8.2 .....</b>	 <b>91</b>
 <b>Readonly classes .....</b>	 <b>92</b>
Untyped and static properties are restricted .....	93
Only readonly class can inherit other readonly classes .....	93
Conclusion .....	94
 <b>Null and false as standalone types .....</b>	 <b>95</b>
Why? .....	96
Gotcha .....	96
 <b>The true type .....</b>	 <b>98</b>
What is true type? .....	98
Limitations .....	99
In conclusion .....	99
 <b>Using constants in traits .....</b>	 <b>101</b>
The problem .....	101

The solution .....	102
Limitations .....	105
In summary .....	107
<b>Disjunctive Normal Form Types .....</b>	<b>108</b>
Limitations .....	109
<b>Dynamic Properties Depreciation .....</b>	<b>110</b>
Allow Dynamic Properties .....	110
Going in PHP 9 .....	111
<b>Fetch properties of enums in const expressions .....</b>	<b>112</b>
Benefits .....	113
<b>Redacting properties in backtraces .....</b>	<b>114</b>
Benefits .....	115
<b>New Random Extension .....</b>	<b>116</b>
<b>The new MySQLi execute_query() method .....</b>	<b>117</b>
<b>Some minor improvements .....</b>	<b>119</b>
Deprecate \${} string interpolation .....	119
Deprecate and Remove utf8_encode and utf8_decode .....	119
Deprecating partially supported callables .....	120
<b>PHP 8.3 .....</b>	<b>121</b>
<b>The json_validate() function .....</b>	<b>122</b>
The signature .....	122

Benefits .....	123
<b>Improved unserialize() error handling .....</b>	<b>124</b>
<b>New methods in the Randomizer class .....</b>	<b>126</b>
The new getBytesFromString() method .....	126
The getFloat() method .....	127
The nextFloat() method .....	128
<b>Fetch class constants dynamically .....</b>	<b>130</b>
<b>Improved Date/Time Exceptions .....</b>	<b>132</b>
Backward compatibility .....	134
<b>Typed Constants .....</b>	<b>135</b>
<b>Readonly amendments .....</b>	<b>137</b>
<b>The #[\Override] attribute .....</b>	<b>141</b>
Benefits .....	144
<b>Arbitrary static variable initializers .....</b>	<b>146</b>
<b>Make unserialize() emit a warning for trailing bytes .....</b>	<b>148</b>
<b>A new mb_str_pad function .....</b>	<b>149</b>
<b>Miscellaneous improvements .....</b>	<b>151</b>
Saner Increment/Decrement operators .....	151
Saner array_sum and array_product functions .....	151

Use exceptions by default in SQLite3 extension .....	152
<b>Deprecations in PHP 8.3 .....</b>	<b>153</b>
Functions with overloaded signatures .....	153
Misc deprecations .....	154
<b>PHP 8.4 .....</b>	<b>155</b>
<b>Property hooks .....</b>	<b>156</b>
The precursor .....	156
What are property hooks? .....	158
Using hooks with interfaces .....	161
Things to note .....	161
In Closing .....	162
<b>The new #[\Deprecated] attribute .....</b>	<b>163</b>
<b>New array methods .....</b>	<b>165</b>
The array_find method .....	165
The array_find_key method .....	166
The array_any method .....	167
The array_all method .....	168
Summary .....	169
<b>Effortlessly parse huge XMLs .....</b>	<b>170</b>
<b>Multibyte equivalents for the trim() function .....</b>	<b>171</b>
<b>A new class for parsing and serializing HTML5 .....</b>	<b>173</b>



<b>Grapheme cluster for <code>str_split</code> function .....</b>	<b>176</b>
<b>Calling methods on a newly instantiated class without parentheses ..</b>	<b>178</b>
<b>New modes for the <code>round()</code> function .....</b>	<b>179</b>
<b>Improvements related to JIT .....</b>	<b>180</b>
<b>Implicitly nullable parameter types will be deprecated .....</b>	<b>181</b>
<b>Separate visibilities for read and write operations on properties .....</b>	<b>182</b>
<b><code>exit()</code> is now a standard function .....</b>	<b>184</b>
<b>A new Enum for rounding modes .....</b>	<b>185</b>
<b>Raising zero to the power of a negative number will give a deprecation warning .....</b>	<b>189</b>
<b>A dedicated class for stream processing .....</b>	<b>190</b>
<b>Session propagation will no longer be done using GET/POST requests .....</b>	<b>191</b>
<b>A new function to efficiently calculate both the quotient and remainder in a single operation .....</b>	<b>192</b>
<b>Improvements to the <code>XMLReader</code> and <code>XMLWriter</code> classes .....</b>	<b>194</b>

<b>Introduction of Lazy Objects .....</b>	<b>195</b>
Creating Lazy Objects .....	195
Handling the State of Lazy Objects .....	196
Lifecycle of Lazy Objects .....	196
A real-world usage of Lazy Objects .....	197
 <b>A new function to allow parsing of multipart/form-data content type for non-POST requests .....</b>	 <b>198</b>
 <b>New additions to BCMath .....</b>	 <b>199</b>
 <b>Some extensions are going away from PHP core .....</b>	 <b>200</b>
 <b>Conclusion .....</b>	 <b>201</b>

# Property hooks

Accessing or setting the value of a class property is a common task in object-oriented programming. There are a few ways to do this in PHP. Let's discuss them first.

## The precursor

Take the following class for example.

```
class User
{
    private string $email;
}
```

As you can tell, we have a private property `$email` in the class. Now, we can define getter and setter methods to read and write the value of the property respectively like so.

```
class User
{
    private string $email;

    public function getEmail(): string
    {
        return $this->email;
    }

    public function setEmail(string $email): void
    {
        $this->email = $email;
    }
}

$user = new User();
$user->setEmail('john@example.com');
echo $user->getEmail(); // john@example.com
```

This is a pretty traditional approach and people have been using it for a long time.

Alternatively, in PHP 8.3, we can shorten this further using the [constructor property promotion](#) like so.

```
class User
{
    public function __construct(public string $email) {}
}

$user = new User('john@example.com');
echo $user->email; // john@example.com
```

This is a pretty neat approach and it's a bit more concise than using the getter and setter methods.

We can use `__get` and `__set` magic methods to achieve the same result as well. But that's very verbose, error-prone, and not friendly for static analysis tools like PHPStan.

PHP 8.4 is going to make this key aspect better by introducing property hooks.

## What are property hooks?

Property hooks, in PHP 8.4, allows you to define custom logic for property access and mutation. This can be useful for a variety of use cases, such as mutation, logging, validation, or caching.

Essentially, property hooks allow you to define additional behavior on class properties mainly using two hooks: `get` and `set`. And this will be individual for certain properties.

### The `set` hook

Here's how we can write a `set` hook for the `$email` property in the previous example.

```
class User
{
    public string $email {
        set (string $value) {
            // validate the email address
            if (!filter_var($value, FILTER_VALIDATE_EMAIL)) {
                throw new InvalidArgumentException(
                    'Invalid email address'
                );
            }
            // Set the value
            $this->email = $value;
        }
    }
}
```

As you can tell, hooks are enclosed in curly braces that come right after the property name. We can then define the hooks inside this code block.

The `set` hook body is an arbitrarily complex method body, which accepts one argument. If specified, it must include both the type and parameter name. Here, we can validate or modify the value of the property before it is set.

So, when a value is set to the `$email` property, the `set` hook will be called and the value will be validated before it is set.

```
$user = new User();  
$user->email = 'example.com'; // This will throw an exception  
  
$user = new User();  
$user->email = 'john@example.com';  
echo $user->email; // john@example.com
```

There's also a **shorthand syntax for defining the `set` hook** using the `=>` operator.

```
class User  
{  
    public string $email {  
        set => strtolower($value);  
    }  
}
```

Here the `$value` is assumed to be the value of the property and the `strtolower` function will be called on it.

## The `get` hook

The `get` hook on the other hand allows you to define custom logic for property access. This can be useful for properties that need to be changed before they are returned to the user.

For instance, if the `User` class has two properties, `$firstName` and `$lastName`, we can define a `get` hook for the `$fullName` property like so.

```
class User
{
    public function __construct(
        public string $firstName, public string $lastName
    ) {}

    public string $fullName {
        get {
            return $this->firstName . " " . $this->lastName;
        }
    }
}

$user = new User('John', 'Doe');
echo $user->fullName; // John Doe
```

As you can tell, the **get** hook does not accept any arguments. It simply returns the value of the property.

So, when a value is accessed from the `$fullName` property, the **get** hook will be called and the value will be returned based on the logic defined in the hook.

There's a **shorthand syntax for defining the **get** hook** using the `=>` operator.

```
class User
{
    public function __construct(
        public string $firstName,
        public string $lastName
    ) {}

    public string $fullName {
        get => string $this->firstName . " " . $this->lastName;
    }
}
```

This is equivalent to the previous example.

## Using hooks with interfaces

The hooks can be specified on interfaces as well.

```
interface Base
{
    // Objects implementing this interface must have a readable
    // $fullName property. That could be satisfied with a traditional
    // property or a property with a "get" hook.
    public string $fullName { get; }
}

class SimpleUser implements Base
{
    // The "get" hook is optional, and if not specified, the
    // property will be readable without a "get" hook.
    public function __construct(public string $fullName) {}
}
```

as you can see, the `$fullName` property is readable without a `get` hook. But if we define a `get` hook for the property, it will be readable with a `get` hook.

## Things to note

There are things to consider while using property hooks.

- Hooks are only available on object properties. So, static properties cannot have hooks.
- Property hooks override any read or write behavior of the property.
- Property hooks have access to all public, private, or protected methods of the object, as well as any public, private, or protected properties, including properties that may have their own property hooks.
- Setting references on hooked properties is not allowed since any attempted modification of the value by reference would bypass a set hook if one is defined.
- A child class may define or redefine individual hooks on a property by redefining the property and just the hooks it wishes to override. The type and visibility of the property are subject to their own rules independently.



So, each hook overrides parent implementations independently of each other.

## **In Closing**

Property hooks are a powerful feature that allows you to customize the behavior of properties in a way that is a lot clearer, concise, and flexible than other approaches. They are especially useful when you want to add custom logic to properties that are read or written by the object.

Although there are two property hooks currently, there's a possibility of adding more in the future which will make property hooks even more powerful!

This is a sample from "PHP 8 in a Nutshell" by Amit D. Merchant.

For more information, [Click here](#).