PHP 8 in a Nutshell

Amit D. Merchant

amitmerchant.com

Table of Contents

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.

```php
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.

```php
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 the 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.

```php
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.

```php
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!