PHP Piscine

Day06 - Video transcript

Bocal **bocal@hive.fi**

Summary:
This document contains transcriptions of the e-learning videos.

Contents

Chapter 1

Attributes and methods

I told you in the previous video that a class can contain attributes and methods. Here we'll see how they can be expressed exactly so that we can use them. First, we're going to talk about attributes, here you see the definition of an attribute called **foo**.

```php
<?php

Class Example {
  public $foo = 0;
    ...
}
?>
```

This is a class variable that we will be able to use from our instances. Here, you will note the keyword **public**. For now, let's not worry about it, we'll come back to it in a next video. For now, all we need to know is that all our instances of our **Example** class will contain an attribute called **foo** which will be initialized by default to **0**.

A small comment about attribute initialization values: they are required to be constants. So you can't put in calculations or function calls, etc. - it has to be a scalar value every time. Be careful with this, it can be a source of errors.
We've already seen the constructor and the destructor, nothing to say about it, nothing has changed.

You can see the definition of a method called **bar**.

```
<?php
Class Example {
        public $foo = 0;

  function bar() {
          print('Method bar called' . PHP_EOL);
          return;
  }
}
?>
```

This method doesn't take parameters and will just display something. This is how you define a method in PHP.

To create an instance of your class, it is still the same principle:

```
<?php
$instance = new Example();

print('$instance->foo:' . $instance->foo . PHP_EOL);
$instance->foo = 42;
print('$instance->foo:' . $instance->foo . PHP_EOL);

$instance->bar();
?>
```

Here we initialize our class and retrieve our instance in the variable **$instance**.
You can see that we now have a new operator, the **->** - which should remind you a lot of the arrow in C. It has exactly the same meaning here. Here, we ask to display the value of the **foo** attribute of our instance with the syntax:

```
$instance variable - Arrow - foo (attribute's name)
```

In our example, we will check its value directly after the class instantiation. Then, using the same syntax, we will assign the value **42** to the attribute **foo** - then, we're displaying the value again. Finally, we call the **bar()** method of my instance of the **Example** class to make sure that everything works properly.

We execute our script:

```
$> php -f Example.class.php
Constructor called
$instance->foo: 0
$instance->foo: 42
Method bar called
Destructor called
```

As expected, at the instantiation of the class, the constructor is called. The value of **foo**, just after the instantiation of the class is 0. Then, remember, my attribute was assigned the value 42 and then this new value was displayed. The value of **foo** has been changed.
Then we called the bar method which displayed a message that you can see here. Finally, once the script is finished, the call to the destructor of our class whose instance is released displays its message.

Whole code of the video:

```
<?php
Class Example {
        public $foo = 0;

        function __construct() {
                print('Constructor called' . PHP_EOL);
                return;
        }

  function __destruct() {
        print('Destructor called' . PHP_EOL);
        return;
  }

  function bar() {
          print('Method bar called' . PHP_EOL);
          return;
  }
}

$instance = new Example();

print('$instance->foo:' . $instance->foo . PHP_EOL);
$instance->foo = 42;
print('$instance->foo:' . $instance->foo . PHP_EOL);

$instance->bar();
?>
```

Chapter 2

PHP accessors

In a previous video, you discovered the notion of encapsulation and therefore, to be able to manage the visibility of your attributes and methods. I can't emphasise enough the importance of this notion which is really central in object-oriented programming. With experience, we quickly realize that there are two ways to use a class. The first one, which is the simplest and the most naive and which reminds a lot of the notion of C structure, is simply to use the class to wrap data together and to be able to move them around as a whole.
In these very simple cases, one realizes that there are rarely any methods associated with this small data packet and one may consider making the attributes public to facilitate manipulation. Even if this is the simplest and most naive case, in practice it is a case that is rarely encountered.
On the other hand, the case we encounter a lot is the case where our class is a real class (with code and data), where the examples are more complicated. In practice, in this case, it is a good habit to put all the attributes in private. That is to say that no attribute is accessible from the outside. Why is this? Simply to be able to keep control over them. The fact that these attributes are private forbids the user of the class to be able to read and write them directly. Of course, it can be interesting to be able to provide the class user with a view on the attributes or a way to influence their value.

That's why we will now discover the notion of **accessor**, what we call **getter** and **setter**. These are very short and simple methods that will allow us, for the **getter**, to expose the value of an attribute to the outside - and, for the **setter**, to be able to allow the outside to modify the value of an attribute.

Let's see an example:

```php
<?php
Class Example {
  $_att = 0;

  public function getAtt() { return $this->_att; }
}
?>
```

Here our class has a private attribute **$_att** that we initialize to 0. Below, we have an accessor, the **getter**, which will allow us to retrieve the value of our attribute. Small comment on the syntax, or the naming convention we use, generally we write **get**, followed by the name of the attribute, capitalized without the underscore. Be careful, this is a convention that it is good to respect since most developers in the world use it. The **getter** is very simple, it will simply return the value of our attribute.

You may be wondering what the point is? Why not allow the value of our attribute to be read directly in public? Simply because if you put the attribute in public, not only do you have access to reading but also to writing - which is something you often don't want to do. Moreover, it gives us the possibility to apply a filter to that value. Let's imagine that we're manipulating money, with a class that represents sums of money in any currency - and when we call an accessor, we can give the value expressed in dollars or euros, and we'll be able to make the conversion directly in order to recover the value that's right. It's an example that doesn't have a lot of interest, but the goal of our **getter** will be to be able, if necessary, to refine the value returned, to present it better so that the user can make better use of it.

We saw the **getter**, which was the easiest, and we're now going to see the **setter**.

```
<?php
Class Example {
  ...
  public function setAtt($v) {
            if ($this->_att + $v > 50)
                    $this->_att = 50;
            else
                    $this->_att = $v;
            return;
    }
}
?>
```

The principle is the same except that the action will not be to read the attribute but to write a value in the attribute. This time, the accessor takes a parameter, **$v**, which represents the value we want to assign to our private attribute. The interest will be, for example, to apply a control on the value that has been sent to us. Does it have a type that is useful to us? Is its value correct in our context? In the proposed example, we say that the attribute **_att** can't be worth more than 50, so I send a value **$v** and I check if the sum of our attribute plus this value is worth more than 50. In this case, I limit my attribute to the value 50, if not, then I assign it the new value. Of course this example has no interest mathematically speaking, the goal is to show you in a few lines what you can do in a **setter**: check the value and do something with it.

Then it's very easy to use! We actually have a very unified approach to our class. When we have a lot of attributes we know that we will always be able to read or write our attribute with the **getter** or **setter** that correspond.

```
<?php
$instance1 = new Example(array('arg' => 42));
$instance2 = new Example(array('arg' => 53));
$instance1->setAtt(30);
print('$instance1->getAtt():' . $instance1->getAtt() . PHP_EOL);
?>
```

In our example, we instantiate the class and modify it. We will see that only **$instance1** is modified and **$instance2** is not. Very normal. Let's execute to be sure:

```
$> php -f Example.class.php
Constructor called
$this->getAtt(): 42
Constructor called
$this->getAtt(): 50
$instance1->getAtt(): 50
Destructor called
Destructor called
```

We have a first constructor, we display the value which was 42. Below that, a second constructor, etc.
We can see that the modification of the first instance is working. Nothing very complicated, but you should know that it is a practice that must absolutely be applied. So, on the other hand, you don't necessarily have to propose an accessor for all the attributes. It is possible to have attributes that are really implementation details of the class and that will never be accessible for reading or writing from the outside. Otherwise, we can also imagine to have some attributes that are only readable, on which we won't be able to modify the value - in this case, we won't propose a **setter** to be able to modify them, only **getters**.

Another good practice will be to systematically use **getter** and **setter** in your code, even internal to the class, to be able to refer to the attributes. Let's imagine that you have a filter to return the value or a check to modify the value of an attribute, when you use them in your code, it's convenient that these filters and modifications apply each time you refer to the attribute, rather than having to do it each time in the code. Don't use read and write calls to your attributes directly in your code, use the **getter** and **setter** that are assigned. You can even imagine having private **getters** or **setters**, so that you only have them for your internal use in your class and they will never be visible or usable from the outside.

Whole code of the video:

```php
<?php
Class Example {
        $_att = 0;

        public function getAtt() { return $this->_att; }

        public function setAtt($v) {
                if ($this->_att + $v > 50)
                        $this->_att = 50;
                else
                        $this->_att = $v;
                return;
        }

        public function __construct(array $kwargs) {
                print('Constructor called' . PHP_EOL);
                $this->setAtt($kwargs['arg']);
                print('$this->getAtt():' . $this->getAtt() . PHP_EOL);
                return;
        }

        public function __destruct() {
                print('Destructor called' . PHP_EOL);
                return;
        }
}

$instance1 = new Example(array('arg' => 42));
$instance2 = new Example(array('arg' => 53));
$instance1->setAtt(30);
print('$instance1->getAtt():' . $instance1->getAtt() . PHP_EOL);
?>
```