PHP Piscine

Day07 - Video transcript

Bocal bocal@hive.fi

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

Contents

Chapter 1

Static

In this video I propose to make a knot in your brain. What we've seen so far has been pretty simple. Now we'll see something a little more subtle. We will develop some notions of vocabulary. I told you yesterday that there are simple methods and static methods. The banal, classic method is part of an instance, it's something dynamic, and the static method belongs to the class, being independent of the instance. The problem with languages, like PHP where sometimes things are a little weird is the use of words incoherently, to designate completely different and sometimes unrelated things.
When I told you about **static** yesterday, I gave you the context of the static method, the class method. Today we will address again, in this video, the keyword **static** but this time to designate something completely different.
The worst part is that, and you will discover this as you develop and learn other programming languages, that all the code of a function, whether it's a simple function, either method, it has what we call a **binding**, and the type of **binding** is determined by how the code is chosen. We will talk on the one hand about the "static binding", which has nothing to do with the keyword **static** which I told you about earlier. A **static binding** means that it is a binding established at compile time, it is hard defined, once and for all.

On the other hand you will find that it also exists for functions a type of binding called a **dynamic binding**. "Dynamic" means that the binding is made at runtime. This is something we can remember because it is valid everywhere (except PHP): what is **static** is related to compilation, code writing, it is "written in stone", and **dynamic** is related to execution, it is performed only at execution.

The methods we've written so far, they all had bindings called **static**, that is, you write the code and the existence of the function, the method, their binding with the class is established once and for all. We will now discover a new use of the keyword **static**, which will allow us to make a dynamic binding. And I too find it strange to use the keyword **static** to designate something dynamic, but that's how it's defined in the PHP language.

We will resume the code of the previous example, used for ”self”. Specifically, we have a class A that has a **foo()** method and a class B that will redefine this **foo()** method.

```php
<?php
Class ExampleA {
        public function foo() {
                print('Function foo from class A' . PHP_EOL);
        }

        public function test() {
                static::foo();
                return;
        }
}

Class ExampleB {
        public function foo() {
                print('Function foo from class B' . PHP_EOL);
        }
}
?>
```

This means that when we call the **foo()** method from a **class A** instance, we get the **foo()** method from **A**, and when we call the **foo()** method from a **class B** instance we will of course obtain the **foo()** method from **B**.

Then we had seen in the previous video that we had the opportunity, using the **test()** method that was inherited, to call the class where the code is defined, in order to choose the function that will be called. I repeat the explanation because it’s a bit complicated, so that we don’t all get confused.
The **test()** method here, as we saw in the previous video, using the keyword **self**, whatever the level of usage in which we use this method, either in **class A** or in **class B**, it will always be the proper class of the **foo()** method, so the **foo()** method where we used **self**, which will be used.

This was static behavior. Now we will use the keyword **static** - sorry about that, again - to talk about a dynamic binding. This time, the correct **foo()** method will be selected only when executing the code.

Let's go back at it. We have two classes, a **class A** and a **class B**. Each of the two classes implements a **foo()** method. We will say that the **foo()** method of **class B**, redefines the **foo()** method of **class A**. If I call the **foo()** method in an **instance A**, clearly the code of the class of the **foo()** method will be executed. Because we redefined the **foo()** method, if I call the **foo()** method in **instance B**, the **foo()** method from the **instance B** will be executed.

Then, as we saw in the previous video, using **self** in the **test()** method, if we call the **foo()** method with **self::foo**, from the **class A** instance, obviously the **foo()** method from **instance A** is called, and we saw that, from the **instance B**, surprisingly, the **foo()** method from **instance A** was called: it was a static behavior, the call was made in relation to the place where the method was defined.

Now replacing the keyword **self** with the keyword **static**, we will get a dynamic binding. Clearly, the choice of word was very unfortunate for the PHP language.
Thanks to this **static** keyword, we will be able to do it in such a way that, calling the **test()** method from **instance B**, which is inherited from **class A**, the code of the **foo()** method from the **class B** will be called.

That is, contrary to the previous example, here, the **foo()** method to be invoked will not be that of **class A** but from the class I'm calling from, which means, the instance of the class that calls the **test()** method.

Let's see this together and run the program.

```
$> php -f Example.class.php
Constructor ExampleA called
Constructor ExampleB called
Function foo from class A
Function foo from class B
Function foo from class A
Function foo from class B
Destructor ExampleB called
Destructor ExampleA called
```

As in the previous example, I construct a **class A** instance, a **class B** instance, which inherits **A** and therefore has an inherited **test()** method. We also redefined the **foo()** method.

When I call the **foo()** method from **class A**, the code of the **foo()** method of **class A** is executed. I do the same in the **class B** instance, and because the **foo()** method has been redefined, the code of the **foo()** method of **class B** is executed. Then things get interesting. From the **class A** instance, if I call the **test()** method making a **static::foo** call, the **foo()** method of **class A** will be executed. And here it is *really* interesting, because if I call the **test()** method which is defined in **class A** and is inherited in **class B**, the call **static::foo** will be binded to the code of the method **foo()** of **class B**.

So to summarize, in our context here, we have the keyword **self** which makes a **static binding**, and the code will be defined once and for all when you write it, that is, the class that uses the keyword **self** will always be the class that will be used to resolve the bind to the method. Now, if we use the keyword **static**, it will be the instance of the class that will allow to determine what code will actually be executed. This time we have a **dynamic binding**. If could say that **self** makes a static connection, and **static** makes a dynamic connection. Of course the terminology is unfortunate, but that's it. Let's not pour our bitterness on PHP, but to pay attention to the advantages that this offers us.

Let's imagine that we have a **class A** that allows us to treat certain data generically, and that at some point we want to manipulate the data in a different way, in a specialized **class B** for that treatment. If we have a collection of as many classes as we can imagine, of type **B** inherits **A**, **C** inherits **A**, **D** inherits **A**, a lot of heterogeneous classes, different specializations of class **A**. Well, all these classes will be able to call a parent class **A** method, having the code of their own instance which will be used to perform data processing. This is close to the meaning of the notion you may have heard in other languages, I am particularly thinking of C++ with its polymorphism by inheritance. It's not exactly the same thing, because we're actually taking the problem the other way around, but the result is this: it allows us to specify a behavior in the generic class, which are used in the specialized class with the specialized class code.

To really understand this concept well, the best thing is to resume this video from the beginning (or read this document again), because it's something pretty subtle, especially since the vocabulary doesn't help us this time. But most of all, experiment on your own. We are doing it in such a way that through today's exercises you will be able to do these manipulations. But try to understand well when the binding is defined. It is written in stone, once and for all, with the word **self**, or or it is defined dynamically using **static**.
I leave you to think about it and we will find you in the next video.

Whole code of the video:

```
<?php
Class ExampleA {
        public function __construct() {
                print('Constructor ExampleA called' . PHP_EOL);
                return;
        }

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

        public function foo() {
                print('Function foo from class A' . PHP_EOL);
        }

        public function test() {
                static::foo();
                return;
        }
}

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

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

        public function foo() {
                print('Function foo from class B' . PHP_EOL);
        }
}

$instanceA = new ExampleA();
$instanceB = new ExampleB();
$instanceA->foo();
$instanceB->foo();
$isntanceA->test();
$instanceB->test();
?>
```