Introdução à Orientação à Objetos

Até aqui fizemos programação estrutural. Agora, orientação a objetos basicamente significa um sistema que é feito de vários objetos. Um exemplo para seu entendimento, um sistema de estoque, em um estoque tem produtos, cada produto desse estoque é um objeto. Então cada objeto é um conjunto de coisas que tem naquele objeto, por exemplo, cor, tamanho, ou seja, vários atributos e funções, características e assim por diante. A partir de agora vamos ver como se aplica essa teoria na prática, como criar, usar um objeto, etc.

Definindo Classes e Objetos

Vamos entender alguns fundamentos da programação orientada a objeto, entender a diferenciação entre uma classe e um objeto.

No conceito de função vocês viram que a partir de criada, usamos a quando quiser, quando referenciamos essa função. Uma classe é exatamente a mesma coisa, vamos definir as propriedades de um objeto. Primeiro definimos as possibilidades do objeto existir, as características e a partir daí sim tornar possível a utilização desse objeto. Essa é a diferença de classe e objeto.

Como exemplo, vamos supor que estamos criando um post. Para isso declaramos uma classe que vai ser responsável por esse post. Para criarmos essa classe, devemos saber quantos likes esse post vai ter, quantos e quais comentários, o autor, quem criou, ou seja, muitas informações e características, além de funções específicas. Veja a declaração dessa classe Post no arquivo a seguir. Declaramos com o termo **class**, isso cria a classe Post. Mas não basta declarar uma classe para ela executar. Com o termo class Post, existe a ideia de um post, só isso por enquanto. Depois disso, declaramos as características dessa classe dentro dela. Essas características podem ser públicas, ou seja, uma característica que conseguimos acessar fora dela. Para isso usamos o termo **public**, conforme podem ver a seguir no arquivo index.php.

Arquivo index.php

```
<?php

    class Post{

        public $likes = 0;

        public $comments = []; //lista de cmentários

        public $author;

    }

?>
```

Agora criada a classe Post, por enquanto ela é inutilizável, não executa, nem tem como definir nada nela. Vamos agora criar o objeto, para isso geralmente usamos o comando new. Depois associamos esse objeto a uma variável, para conseguirmos ter acesso a esse objeto. Para definirmos as propriedade num objeto usamos seta (->) . Veja a seguir no index.php:

Arquivo index.php

```
<?php

class Post{

	public $likes = 0;

	public $comments = [];

	public $author;

}

//Primeiro objeto:

$post1 = new Post();

$post1 -> likes = 3;  //aqui definimos que o objeto vai ter 3 likes.

//Segundo objeto:

$post2 = new Post();

$post2 -> likes = 10; //aqui definimos que vai ter 10 likes

echo "Post 1: ".$post1->likes."<br/>";

echo "Post 2: ".$post2->likes;

?>
```

Esse exemplo foi só para vocês entenderem a diferença entre classe e objeto. Observem que o modelo da classe, foi aplicada nos 2 objetos. Pega as propriedade originais do modelo que é a classe e importa para esses objetos.

Agora se perguntar para vocês qual é o objeto do post1 ? vocês tem que saber que é a variável $post1. Que ela é um objeto que foi construído a partir de um modelo, que esse modelo é a classe Post. Entendendo esses conceitos iniciais, vamos agora entender internamente como isso funciona.

Definindo Métodos e Propriedades

Vamos agora entender o que são métodos, propriedade ou atributos como queira chamar, ou ainda no PHP 7.4 o que são os typed properties (propriedades tipadas).

As propriedades foi o que criamos na classe no exemplo anterior. Elas são as características que uma classe vai ter, que por consequência quando for criado o objeto, esse também vai ter essas características. Existem vários tipos de atributos ou propriedades em uma classe. No exemplo anterior criamos propriedades publicas (public), que significa que podemos de fora dessa classe alterar, acessar informações, é acessível publicamente. Temos ainda as propriedades **protected** (protegidas) e **private** (privada), ambas essas propriedades elas não são acessíveis do lado de fora da classe.

Observem alterações no arquivo index.php:

Arquivo index.php

```
<?php

class Post{

//Alteramos de publica para privada

	private $likes = 0;

	public $comments = [];

	public $author;

}


$post1 = new Post();

$post2 = new Post();


echo "Post 1: ".$post1->likes."<br/>";

echo "Post 2: ".$post2->likes;

?>
```

Observem agora, executando esse arquivo que vai dar o seguinte erro:

Fatal error: Cannot access private property Post::$likes in C:\wamp\www\php\index.php on line 12

Esse erro é devido à propriedade $likes ser private, não permitiu acesso de fora.

Agora vocês devem estar se perguntando, mas qual é o sentido de se ter uma propriedade que não se consegue acessar? O sentido é de usar essa propriedade somente internamente a classe. Quando não queremos dar acesso à determinada propriedade para o mundo exterior assim dizer, protegendo a classe de interferências externas. O uso da propriedade protected, que é muito similar a private. Essa diferença será vista em exemplos mais adiante, que possibilitará um melhor entendimento.

Agora vamos falar sobre métodos. O que são métodos? Como foi explicado o objeto tem características, que são as propriedades e ele também pode executar coisas, funções, executar tarefas através do próprio objeto. Por exemplo, vamos criar um método específico para aumentar a quantidade de likes, para adicionar um like. Como vamos fazer esse processo? Para isso vamos à classe, é na classe que se altera tudo, e nos objetos só usamos essas coisas. Todo método vai ter as propriedades public, protected e private do mesmo jeito. Por exemplo, vamos criar o método aumentarLike(), mas observem que o código a seguir vamos fazer dentro da classe Post no arquivo index.php.

Arquivo index.php

```
<?php

class Post{

	public $likes = 0;

	public $comments = [];

	public $author;

	//método aumentarLike()

	public function aumentarLike(){

	//$post1->likes++;  //obs que não podemos fazer assim para pegar a variável $likes, //pois não existe $post1 dentro da classe... para isso usamos o $this

	$this->likes++;

	}

}

$post1 = new Post();

//aqui da mesma forma que fizemos para acesso as propriedade, fizemos para os métodos:

$post1->aumentarLike(); //vai aumentar a qtidade de likes de post1.

$post2 = new Post();


echo "Post 1: ".$post1->likes."<br/>";
```

```
echo "Post 2: ".$post2->likes;

?>
```

This em inglês quer dizer isso, que está próxima, ela mesma. Então na programação quanto tem this quer dizer que está se referenciando ao próprio item. Como em $this->likes++; a própria variável $likes. Então se querguntar quem é o this de $post1 = new Post(); ? O próprio $post1. Quem é o this de $post2 = new Post() ? O próprio $post2. Então cada um desses dois tem o seu próprio this.

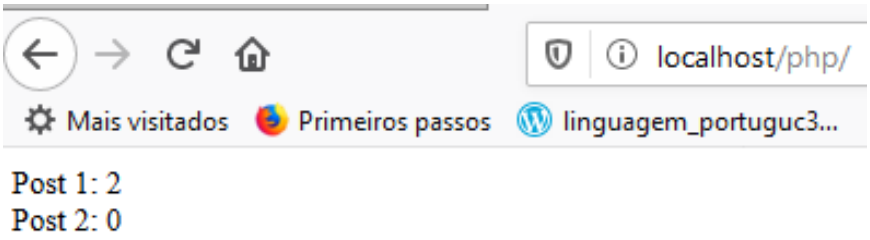
No exemplo, os likes dos próprios itens que acessarmos vão ser aumentados.

Executando o index.php observem que vai aumentar o número de likes para o objeto $post1 ao executar esse método, conforme mostra a figura a seguir:

Observem que antes o valor estava zerado para os dois objetos, no entanto executou e adicionou uma like. Se repetirmos no código o método:

```
$post1->aumentarLike();

$post1->aumentarLike();
```

e atualizar, no site vai aumentar mais uma like:



Se agora quisermos aumentar uma like para o objeto $post2, é só colocar no código:

```
$post2->aumentarLike();
```

Agora por que não podemos fazer direto esse incremento em um no objeto, por exemplo:

```
$post1 = new Post();
```

$post1->likes++;

Por que, quando usamos um sistema real, mandamos requisições para um servidor, para um banco de dados, que temos que atualizar as informações. E essas informações tem que estar acessíveis a todos que acessarem esse post. Então por isso criamos métodos, para uma série de funções, que não vamos fazer manualmente toda hora. Lembram-se do princípio de funções, quando precisamos executar um grupo de código mais de uma vez, então criamos uma função.

Com isso percebem que classes têm esses itens que são métodos, funções e propriedades.

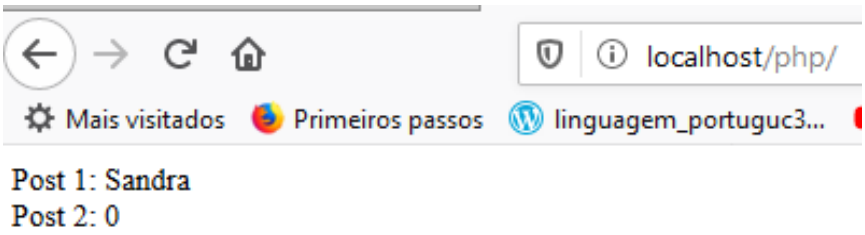
Typed Properties (7.4)

Propriedades típadas, recurso que veio junto com o PHP 7.4. Significa que em uma classe a partir disso, temos como proteger uma propriedade, para receber somente um tipo de informação. Por exemplo, observem a seguir a alteração no código do arquivo index.php, onde está destacado em vermelho:

```
<?php

class Post{

        public $likes = 0;

        public $comments = [];

        public $author;

        //método aumentarLike()

        public function aumentarLike(){

        //$post1->likes++;  //obs que não podemos fazer assim para pegar a variável $likes, //pois não existe $post1 dentro da classe... para isso usamos o $this

        $this->likes++;

        }

}


$post1 = new Post();

$post1->likes = 'Sandra';

$post2 = new Post();


echo "Post 1: ".$post1->likes."<br/>";
```

```
echo "Post 2: ".$post2->likes;

?>
```

Após executar esse arquivo mostrará assim:



Ou seja, a $likes recebeu uma string. Com a propriedade **typed properties** temos como proteger essa variável para receber um tipo específico de dados. Para isso basta irmos à classe Post e entre **public $likes**, colocar **int**. Assim receberá somente valore do tipo inteiro. Veja no arquivo index.php essa aplicação:

```
<?php

class Post{

        public int $likes = 0;

        public $comments = [];

        public $author;

        //método aumentarLike()

        public function aumentarLike(){

        //$post1->likes++;  //obs que não podemos fazer assim para pegar a variável $likes, //pois não existe $post1 dentro da classe... para isso usamos o $this

        $this->likes++;

        }

}


$post1 = new Post();

$post1->likes = 'Sandra';

$post2 = new Post();
```

```
echo "Post 1: ".$post1->likes."<br/>";

echo "Post 2: ".$post2->likes;

?>
```

Observem ao executar esse código aponta um erro:

Parse error: syntax error, unexpected 'int' (T_STRING), expecting variable (T_VARIABLE) in C:\wamp\www\php\index.php on line 3.

Por que deu esse erro? Por que essa variável está recebendo uma string ($post1->likes = 'Sandra';), ela está definida como inteira, não aceita string, pois ela esta protegida com typed properties. Se por exemplo trocarmos $post1->likes = 'Sandra' por $post1->likes = 15; ai sim aceita esse valor por ser do tipo inteiro. Isso deixa a classe muito mais confiável, um recurso muito bom do php 7.4.

Do mesmo jeito podemos proteger as outras variáveis na classe, por exemplo:

```
<?php

class Post{

        public int $likes = 0;

        public array $comments = [];

        public string $author;


        public function aumentarLike(){

        $this->likes++;

        }

}
```

Método Construtor

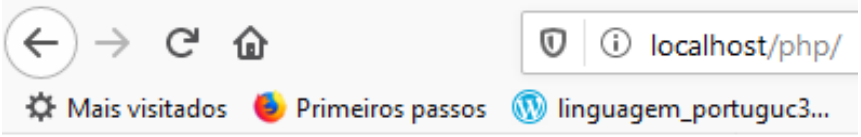
Uma classe tem a possibilidade de usar alguns métodos específicos que são meio predefinidos, pré-programados para rodar automaticamente, um deles é o chamado construtor. Como criamos esse método construtor? Construímos assim:

```
 public function __construct(){

}
```

Para que serve? Observem a aplicação no arquivo **index.php**:

```
<?php
class Post{
	public int $likes = 0;
	public array $comments = [];
	public string $author;
	public function __construct(){
		echo 'Teste';
	}

	public function aumentarLike(){
		echo 'ABC';
	$this->likes++;
	}
}
$post1 = new Post();
$post2 = new Post();

echo "Post 1: ".$post1->likes."<br/>";
echo "Post 2: ".$post2->likes;
?>
```

Agora vamos executar esse arquivo index.php com essas alterações e observem o que mostra:



TesteTestePost 1: 0
Post 2: 0

Vocês devem estar se perguntando como foi que essas informações apareceram TesteTeste. Pois não estamos usando o __construct(), percebem, não referenciamos ele fora da classe. Isso porque o construtor é um método, quando criamos ele, executa toda vez que criamos um objeto novo. Percebem o nome construtor, que constrói então ele é executado toda vez que um objeto é criado objeto. Então percebem que o construtor foi rodado em:

$post1 = new Post();

$post2 = new Post();

Por isso, a saída de TesteTeste. E percebam que ABC não aparece por que não foi utilizado.

Agora, para que serve esse construtor? Sempre que você precisa executar alguma coisa no momento em que o objeto é criado, usa o construtor. Vamos ver daqui em diante vários exemplos práticos de uso de construtor. Um deles é usado para definir as propriedades iniciais do objeto, ou seja, criar o objeto com as informações corretas. Por exemplo:

$post1 = new Post(); //aqui criamos o objeto, e entre esse intervalo de criar e adicionar propriedades, ele ficou com zero.

$post1->likes = 10;

Então vamos ver como criar objeto já com as informações corretas. Para isso criamos parâmetros. Vamos fazer com que o primeiro parâmetro é a quantidade de likes, iguais a 25 observem onde está em vermelho no código. E como recebe essa informação desse parâmetro no código? Recebe a informação no __construct().

Arquivo index.php

```
<?php

class Post{

        public $likes = 0;

        public $comments = [];

        public $author;

        public function __construct($qtLikes){

                $this->likes = $qtLikes;
```

```
        }


        public function aumentarLike(){
                echo 'ABC';
        $this->likes++;


        }
}


$post1 = new Post(20); //Então, qdo for criado o obj, automaticamente está enviando a
                //qtdade de likes inicialmente que o objeto tem.
$post2 = new Post();


echo "Post 1: ".$post1->likes."<br/>";
echo "Post 2: ".$post2->likes;
?>
```

Observe, ao executar o código do arquivo index.php como está acima, vai dar um erro:

Warning: Missing argument 1 for Post::__construct(), called in C:\wamp\www\php\index.php on line 19 and defined in C:\wamp\www\php\index.php on line 6

Por quê ? Esse erro deu devido o segundo objeto $post2, que está sem parâmetro, então o public function __construct($qtLikes) está esperando desse objeto também o parâmetro.

$post2 = new Post();

Como resolver isso ? Podemos tornar o __construct() um valor padrão, por exemplo:

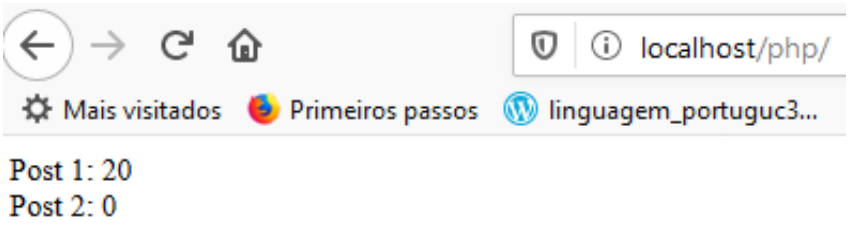
public function __construct($qtLikes = 0), senão mandar valor como parâmetro ele vai assumir 0. Tornamos o parâmetro $qtLikes = 0 opcional.

Veja no código como foi feito essas alterações:

```
<?php
class Post{
```

```
        public $likes = 0;
        public $comments = [];
        public $author;
        public function __construct($qtLikes = 0){
                $this->likes = $qtLikes;
        }

        public function aumentarLike(){
                echo 'ABC';
        $this->likes++;
        }
}
$post1 = new Post(20);
$post2 = new Post();
echo "Post 1: ".$post1->likes."<br/>";
echo "Post 2: ".$post2->likes;
?>
```

Agora se executarmos o index.php não vai dar mais esse erro da falta de parâmetro no objeto como podem ver:



Isso ocorre porque, antes mesmo de criar o objeto, o __construct($qtLikes = 0) já gerou, alterou e construiu o objeto com as propriedades corretas.

Esse método __construct() é o primeiro método executado quando vamos criar um objeto novo. Então o construtor serve pra isso, o que precisa executar assim que o objeto é criado executamos

e fazemos dentro do construtor. Quer seja auxiliando no preenchimento de informações ou fazendo alguma outra coisa específica. Por exemplo, quer saber o id desse post, então vamos colocar um variável para ter esse id, public int $id.

Veja a seguir a definição para saber o id desse Post:

```
<?php

class Post{

	public int $id;

	public $likes = 0;

	public $comments = [];

	public $author;

	public function __construct($postId){

		$this->id = $postId;

		//Consultar bco de dados para pegar informações do Post $id...

		//Só um mero exemplo, certo alunos ?

		$this->likes = 12;

	}


	public function aumentarLike(){

		echo 'ABC';

	$this->likes++;


	}

}


$post1 = new Post(1);

$post2 = new Post(1);

echo "Post 1: ".$post1->likes."<br/>";

echo "Post 2: ".$post2->likes;
```

?>

Percebam que foi feita ações variadas, como preencher variável e consultar banco de dados num construtor.

Entendendo Encapsulamento

Conceito de encapsular uma classe nada mais é do que proteger as propriedades de acessos externos ou então de modificações desnecessárias, modificações que vão prejudicar o funcionamento daquele objeto. Existe quase um **sinônimo** hoje em falar **encapsulamento** ou então falar em criar o **set/get** de determinada coisa. Vamos fazer no código a seguir:

```
//1ª passo:

$post1 = new Post();

$post1->author = 'Sandra';


$post2 = new Post();

$post2->author = 'Fulano';


//2ª passo:

echo "Post 1: ".$post1->likes." likes = ".$post1->author."<br/>";

echo "Post 2: ".$post2->likes." likes = ".$post2->author."<br/>";
```

Até aqui podem ver nada de mais no código anterior. Mas no conceito de encapsulamento recomenda não fazer o que vou colocar na escrita em vermelho a seguir:

```
$post1->author = 'Sandra';

echo "Post 1: ".$post1->likes." likes = ".$post1->author."<br/>";
```

Recomenda fazermos assim:

```
$post1->setAuthor('Sandra');

echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
```

Agora devemos criar esses dois métodos. Vejam no index.php

```php
<?php
class Post{
	public int $likes = 0;
	public array $comments = [];
	public string $author;

	public function aumentarLike(){
		$this->likes++;
	}
	//Esse método vai receber um nome
	public function setAuthor($n){
		$this->author = $n;
	}
	//Segundo médodo
	public function getAuthor($n){
		return $this->author;
	}
}
//1ª passo:
$post1 = new Post();
$post1->setAuthor('Sandra');

$post2 = new Post();
$post2->setAuthor('Fulano');

//2ª passo:
echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
```

```
echo "Post 2: ".$post2->likes." likes = ".$post2->getAuthor()."<br/>";

?>
```

Vocês devem estar se perguntando por que precisa usar um método específico para setar uma informação $post1->setAuthor('Sandra'), se podemos fazer o que o próprio método está fazendo $this->author = $n .

Por que fazendo assim $post1->setAuthor('Sandra') permite tratar alguma coisa naquela informação específica. Um exemplo se fizermos assim:

```
$post1->author = 'sandra';
```

Vejam que no nome sempre colocamos a primeira letra em maiúscula, o que nesse código não possibilitaria de fazer automaticamente com uso de funções nativas por exemplo. O correto é ter a informação corretamente dentro do objeto. Então fazendo da forma que está na linha acima não temos um controle dessa variável. Quando usamos setAuthor nos possibilita ter esse controle, por exemplo:

No método setAuthor na variável que recebe o nome usamos a função pra transformar a primeira letra do nome em maiúscula como mostra em vermelho a seguir:

```
public function setAuthor($n){
        $this->author = ucfirst($n);
    }
```

Veja no index.php a seguir com essa aplicação:

```
<?php
class Post{
    public int $likes = 0;
    public array $comments = [];
    public string $author;

    public function aumentarLike(){
        $this->likes++;
    }
    //Esse método vai receber um nome
```

```
	public function setAuthor($n){
		$this->author = ucfirst($n);;
	}
	public function getAuthor($n){
		return $this->author;
	}
}
//1ª passo:
$post1 = new Post();
$post1->setAuthor('Sandra');

$post2 = new Post();
$post2->setAuthor('Fulano');

//2ª passo:
echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
echo "Post 2: ".$post2->likes." likes = ".$post2->getAuthor()."<br/>";
?>
```

Para cada uma das propriedades vamos criar dois métodos: um **set** outro **get**. Um para setar a informação e outro para pegar essa informação. Junto disso, vamos à propriedade public string $author e transformar ela numa propriedade privada. Porque o que é público é somente o método public function getAuthor($n), ou seja a variável $author só pertence dentro da classe. Então alteramos para private string $author. A partir de agora a única forma que temos de alterar essa variável $author é através do setAuthor:

```
public function setAuthor($n){
		$this->author = ucfirst($n);;
}
```

Através do setAuthor que possibilita fazermos as devidas verificações. Por exemplo, o nome do autor tem que ter 3 ou mais caracteres, senão não troca o autor:

```
public function setAuthor($n){
	if(strlen($n) >=3){
		$this->author = ucfirst($n);
	}
}
```

Então, vejam que isso tudo já são sistemas de proteção. Observem o parâmetro do primeiro objeto não satisfaz o critério do tamanho do nome, se executar vai apontar erro, ou seja, está tentando acessar o

```
echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
```

Arquivo index.php

```
<?php
class Post{
	public int $likes = 0;
	public array $comments = [];
	private string $author;

	public function aumentarLike(){
		$this->likes++;
	}
	public function setAuthor($n){
		$this->author = ucfirst($n);;
	}
	public function getAuthor($n){
		return $this->author;
	}
```

```
}

$post1 = new Post();
$post1->setAuthor('Sa');

$post2 = new Post();
$post2->setAuthor('Fulano');

echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
echo "Post 2: ".$post2->likes." likes = ".$post2->getAuthor()."<br/>";
?>
```

Para resolver esse erro existem 2 formas:

Primeira, setamos um valor padrão para a variável assim

```
private string $author = '';
```

que a partir disso roda normal, só não mostra o autor devido não valida por ser o nome somente com 2 caracteres.

O segundo, deixamos assim a propriedade private string $author; e vamos ao método:

```
public function getAuthor($n){
        return $this->author ?? '';
        //ou seja, se tiver usa, ou senão usa o valor padrão
}
public function getAuthor($n){
        return $this->author = 'Visitante';
}
```

Veja o código completo a seguir:

Arquivo index.php

```php
<?php
class Post{
	public int $likes = 0;
	public array $comments = [];
	private string $author;

	public function aumentarLike(){
		$this->likes++;

	}
	//Esse método vai receber um nome
	public function setAuthor($n){
		$this->author = ucfirst($n);;
	}
	public function getAuthor($n){
		return $this->author ?? '';
	}
}
$post1 = new Post();
$post1->setAuthor('Sa');

$post2 = new Post();
$post2->setAuthor('Fulano');

echo "Post 1: ".$post1->likes." likes = ".$post1->getAuthor()."<br/>";
echo "Post 2: ".$post2->likes." likes = ".$post2->getAuthor()."<br/>";
?>
```

Então, tanto para setar ou pegar a informação, usando encapsulamento, temos acesso a fazer a manipular certa. Temos acesso para deixar o objeto do jeito que tem que ser protegido. Mesmo que mande informações erradas vai filtrar. Recurso esse recomendado a usar em todas as propriedades. Cria uma propriedade, cria o set e o get dela. Manter isso como costume.

Método Estático

Método estático é um método que você vai fazer dentro da sua classe, que vai ser independente. Ou seja, pode ser usado unicamente em uso externo. Exemplo, vamos fazer uma classe Matematica, responsável por matemática, então tudo que for responsável por matemática vai usar essa classe. Vamos criar uma função dentro dessa classe, chamada somar. Lembrando que toda função dentro de uma classe é chamada de método. Então vamos ter o método somar. Para usarmos esse método, primeiro vamos ter que extanciar a classe criando um objeto. Normalmente fizemos conforme está no código a seguir do arquivo index.php:

```
<?php
    class Matematica{
        public function somar($x, $y){
            return $x + $y;
        }
    }
    $n = new Matematica();
    echo $n->somar(10, 20);
?>
```

Depois de executar e retornar esses valores somados, agora vamos transformar esse método em um método estático. Fazendo isso não vamos mais precisar criar um objeto para depois utilizar a função ou método somar(), ou seja, vamos usar esse método externamente sem criar um objeto. Isso chama-se função estática. Para isso vamos ao método somar() e colocamos a termo **static** entre public function. Observem as alterações no código a seguir:

```
<?php
    class Matematica{
        public static function somar($x, $y){
            return $x + $y;
```

```
        }
    }
    //a seguir referenciamos a classe...
    echo Matematica::somar(10, 20); //
?>
```

Observem assim, que colocamos o termo **static** entre public e function já se torna uma função estática. Depois para usar essa função estática temos que referenciar a classe a qual essa função faz parte. Como foi feito em echo **Matematica::somar(10, 20)**, ou seja o nome da classe seguido por dois pontos (**::**) e o nome da função que quer usar. Está indicando conforme exemplo anterior, que na classe Matematica tem uma função estática e use-a. Os dois pontos representa isso, acesse uma propriedade, uma função ou um método que está dentro da classe referenciada.

Sendo uma função estática conseguimos usá-la sem precisar criar um objeto.

Veja a seguir um exemplo feito numa propriedade:

```
<?php
    class Matematica{
        public static string $nome = 'Sandra';
        public static function somar($x, $y){
            return $x + $y;
        }
    }
    //acesso a propriedade $nome
    echo Matematica::$nome;
?>
```

Observem se tirarmos o termo static do código, não temos como acessa os dados usando dessa forma echo Matematica::**$nome**; por não ser uma propriedade estática.

Exercício Prático

Fazer por primeiro uma classe, que vai ser chamada de Calculadora, a seguir está o código de uso dela, a implementação que fica num arquivo chamado **index.php**:

```
<?php

require 'calculadora.php';

$calc = new Calculadora();

$calc->add(12);

$calc->add(2);

$calc->sub(1);

$calc->mult(3);

$calc->div(2);

$calc->add(0.5);


echo "Total: ".$calc->total();

$calc->clear();

?>
```

Observem como é o funcionamento desse código no index.php. Primeiro foi adicionado 12 + 2, depois subtraiu por 1, depois multiplicou por 3, depois dividiu por 2 e por último adicionou mais 0.5 e após mostra esse total na tela. Se vocês usarem esses mesmos valores o resultado na tela será **Total: 20**

E no arquivo **calculadora.php**, vocês devem fazer a própria classe Calculadora.