PHP – Orientação a Objetos.

\*Um dos pilares da orientação a objeto é abstração, ou seja, abstrair o que é necessário para nosso sistema.

\*O nome orientação a objetos vem do fato de que, tenho uma classe, a partir dessa classe posso criar vários objetos, e a partir desses objetos eu monto toda minha programação, monto meu sistema, ou seja, ela deixa de ser orientada a funções (que é a programação procedural), e agora ela passa a ser orientada aos objetos que tenho no meu sistema, por isso, o nome de programação orientada a objetos.

\*Os dados de uma classe são chamados de atributos, são os dados que serão utilizados pelos objetos, são atributos.

\*A classe é uma forma, ou uma planta e nela você cria os objetos, com isso você pode ter vários objetos com valores diferentes, LEMBRE-SE Não na classe, mais sim no objeto, em cada objeto agente define os seus valores.

\*Uma função que está dentro de uma classe é chamada de método, e ele não é copiado para cada instância ou para cada objeto, ele fica na classe, dentro dele temos a referência para o objeto que chamou esse método, com isso conseguimos saber os atributos de qual objeto eu estou alterando, no exemplo apresentado, eu estou alterando o atributo saldo DA CONTA QUE CHAMOU ESSE MÉTODO E NÃO DE TODAS AS CONTAS.

Public function sacar(float $valorASacar){

If ($valorASacar > $this ->saldo){

echo “saldo indisponível”;

} else {

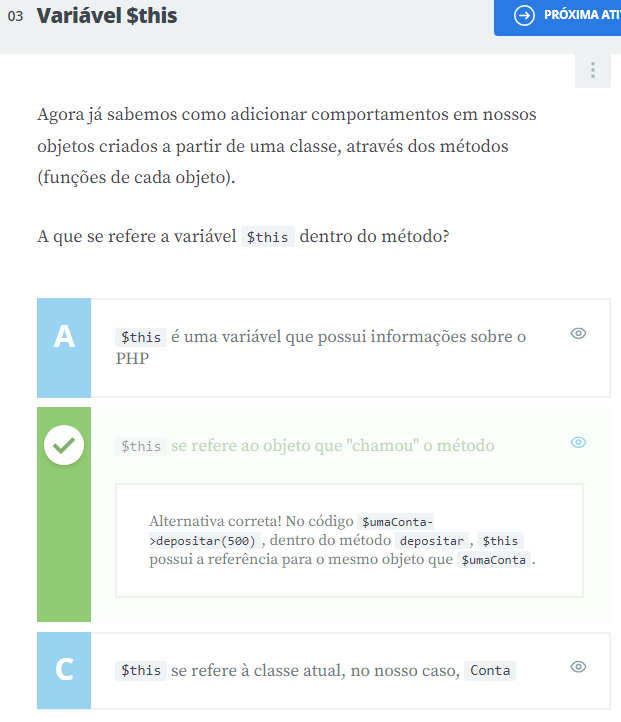
$this->saldo -= $valorASacar;

}

}

* Que variável **$this** tem a referência para o objeto que chamou o método atual

\*Recomendação geral onde tenha apenas propriedades sempre privadas e só os métodos sejam públicos.



Getters e setters **são usados para proteger seus dados, especialmente na criação de classes**. Para cada instância de variável, um método getter retorna seu valor, enquanto um método setter o define ou atualiza.

Os métodos que devolvem o valor de um atributo da classe são chamados de **getters**, e é muito comum que sejam escritos na forma getSaldo(), ou seja, get mais o nome do atributo da classe. Alguns programadores preferem nomear os métodos e atributos de maneira mais homogênea - se estão programando em português, evitam usar termos em inglês. Nesse caso, se tivéssemos uma propriedade $name, poderíamos ter também um método getName().

Já os métodos **setters** definem os valores de atributos privados. Porém, que diferença faz mantermos os atributos privados e criarmos setters para eles? A resposta é que os métodos setters podem ter algum tipo de validação, por exemplo definindo que o $nomeTitular tenha pelo menos 5 caracteres e um espaço (para recebermos nome e sobrenome), ou validar um CPF de modo a garantir que ele seja adequado.

Portanto, os setters são interessantes para termos certeza de que os valores passados às nossas instâncias façam sentido, impedindo, por exemplo, que seja passado um número de telefone no $cpfTitular.

Assim, conhecemos até o momento dois pilares fundamentais da orientação a objetos:

* **abstração**, na qual trazemos o que é necessário do "mundo real" ao nosso sistema
* **encapsulamento**, no qual só devemos expôr o que for necessário da nossa aplicação

Após a execução, vemos que os atributos $cpfTitular e $nomeTitular estão nulos. Entretanto, não existe Conta sem titular ou CPF, indicando que, nesse momento, nossa conta está em um estado inconsistente. Nosso objetivo é, no momento da criação de uma Conta, executarmos um código garantindo que teremos um $cpftitular e um $nomeTitular, e também definindo o $saldo como 0.

O evento de criar o objeto de uma classe faz a execução de um dos chamados *métodos mágicos* do PHP, que é o **método construtor**. Em algumas linguagens o construtor não é um método, mas no PHP é, e é definido como qualquer outro, exigindo apenas um nome especial: \_\_construct().

Sempre que criarmos uma Conta, o método \_\_construct() será chamado. Para testarmos isso, faremos um echo da mensagem "Criando uma nova conta" no corpo desse método.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo = 0;

public function \_\_construct()

{

echo "Criando uma nova conta" . PHP\_EOL;

}

//...COPIAR CÓDIGO

Ao executarmos banco.php, a mensagem "Criando uma nova conta" será exibida no console. Da mesma forma, se criarmos quatro contas, a mensagem será exibida quatro vezes, mesmo que criemos somente uma variável.

Porém, ainda continuaremos com o problema de termos uma conta em estado inconsistente. Queremos que, no momento da criação de uma conta, seja necessário informar o $cpf e o $nomeTitular. Isso é possível definindo que o construtor precisa receber uma string $cpfTitular e uma string $nomeTitular. No corpo do método, faremos com que os atributos da nossa instância recebem os valores passados por parâmetro.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

public function \_\_construct(string $cpfTitular, string $nomeTitular)

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

$this->saldo = 0;

}COPIAR CÓDIGO

Com isso, inicializaremos os atributos da classe Conta com os parâmetros passados pelo construtor. Inclusive, no PhpStorm, a chamada de new Conta() será realçada em amarelo, indicando que existem parâmetros obrigatórios. Agora poderemos passar diretamente o CPF 123.456.789-10, e o nome Vinicius Dias na criação de $primeiraConta, além de removermos a chamada de defineCpfTitular().

A partir do momento em que uma conta é criada, segundo nossa regra de negócios, não deve mais ser possível trocar o titular. Sendo assim, poderemos remover da definição da classe os métodos defineNomeTitular() e defineCpfTitular(), afinal, segundo o que aprendemos sobre encapsulamento, só devemos expôr para o mundo exterior o que é necessário.

Para garantirmos que os valores estão sendo passados corretamente, faremos um $echo de $primeiraConta->recuperaNomeTitular() e de $primeiraConta->recuperaCpfTitular(), adicionando . PHP\_EOL ao final de cada chamada para quebrarmos linha entre os retornos.

require\_once 'src/Conta.php';

$primeiraConta = new Conta('123.456.789-10', 'Vinicius Dias');

$primeiraConta->deposita(500);

$primeiraConta->saca(300); // isso é ok

echo $primeiraConta->recuperaNomeTitular() . PHP\_EOL;

echo $primeiraConta->recuperaCpfTitular() . PHP\_EOL;

echo $primeiraConta->recuperaSaldo() . PHP\_EOL;COPIAR CÓDIGO

Vinicius Dias

123.456.789-10

200

Se tentarmos criar uma $segundaConta seremos obrigados a passar um CPF e um nome do titular - inclusive, a execução de um código sem esses parâmetros retornará um erro.

$segundaConta = new Conta();COPIAR CÓDIGO

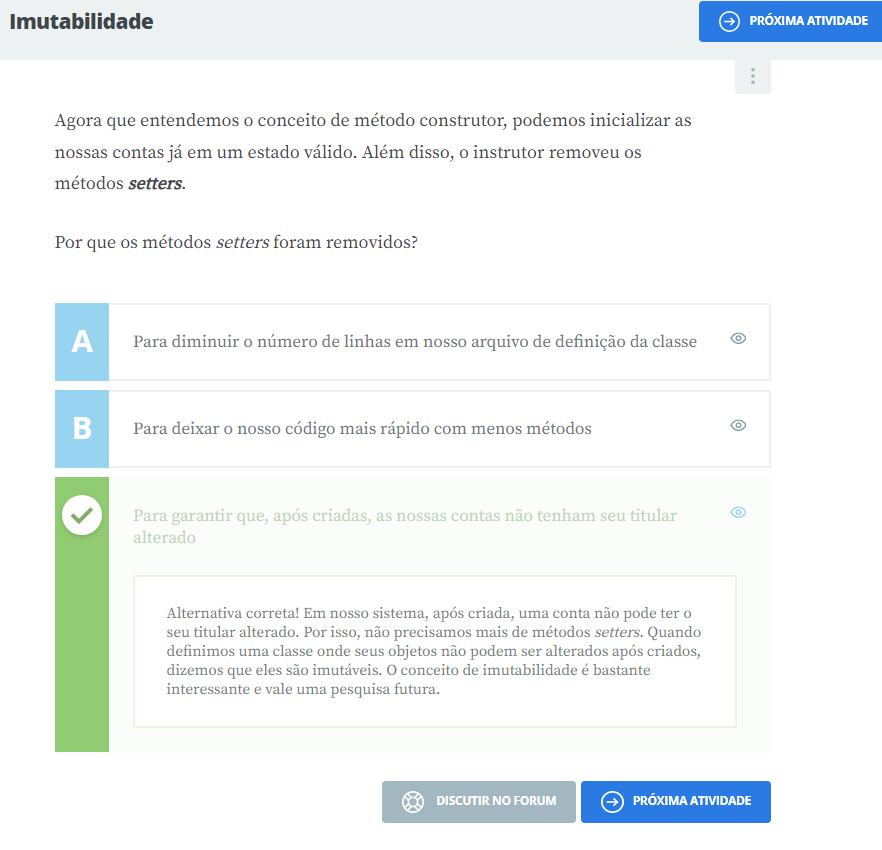
PHP Fatal error: Uncaught ArgumentCountError: Too few arguments to function Conta::\_\_construct(), 0 passed

Corrigiremos isso passando os argumentos necessários para nossa instância.

$segundaConta = new Conta('698.549.548-10', 'Patricia');COPIAR CÓDIGO

Perceba que não estamos passando um $saldo, já que, segundo nossa própria regra de negócios, ele é inicializado como 0 diretamente no construtor. É importante deixar claro que não devemos executar muitas regras de negócio no construtor, mas apenas inicializar a nossa instância, que é exatamente o que estamos fazendo.

Temos então um código que é executado sempre que uma nova conta é criada, o que é muito importante para inicializarmos atributos ou se quisermos fazer alguma validação da classe. Ainda nesse capítulo entenderemos como fazer alguns tipo de validações sem prejudicar a organização do nosso código.



Entendemos o que são membros estáticos. São membros da classe em si, e não de cada instância (objeto).

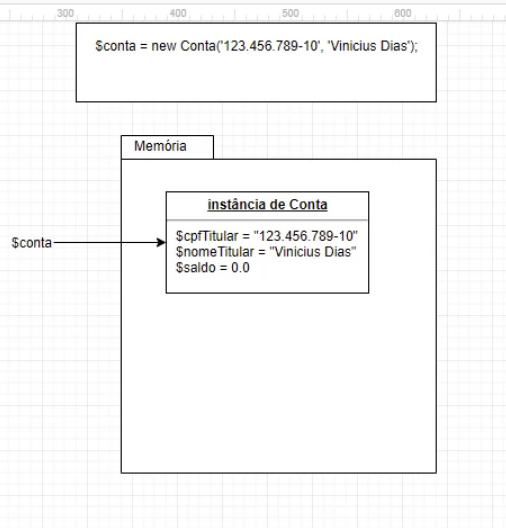
Apesar de parecerem úteis, podem ser bastante danosos, se escritos indiscriminadamente.

Nos treinamentos sobre testes aqui da Alura, este problema fica mais claro.

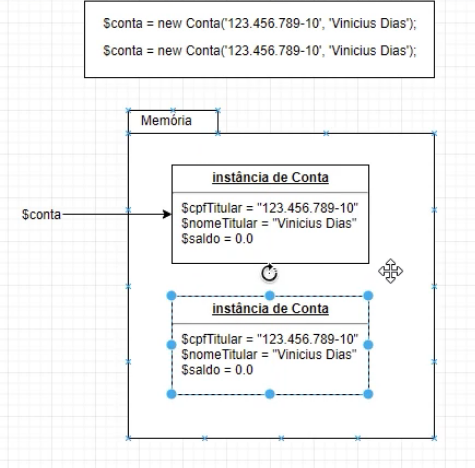
Apenas escreva métodos estáticos quando eles realmente forem necessários e fizerem sentido (é raro).

----------------------------------------------------------------------------------------------------

Observando o diagrama abaixo, temos um código que instancia uma nova conta e gera, como resultado, uma variável $conta que aponta para um endereço da memória onde está armazenada uma instância de Conta com os valores que foram definidos.



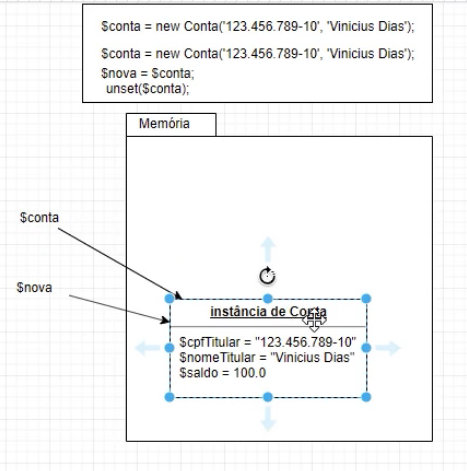
Se criarmos uma nova $conta, com os mesmos dados ou não, teremos uma nova instância.



Porém, repare que estamos atribuindo o endereço de memória da nova conta à mesma variável $conta. Ou seja, estamos sobrescrevendo esta variável, que passa a apontar para a segunda instância. Enquanto isso, a primeira instância está "abandonada", sem nenhuma referência que aponte a ela. Ainda que o dado continue existindo na memória do computador, não conseguiremos mais acessá-lo.

O PHP tem um mecanismo interessante chamado **coletor de lixo**, ou garbage collector, que também existe em outras linguagens. Durante a execução do programa, o interpretador do PHP chamará o garbage collector para verificar todos os dados que estão sem nenhuma referência, removendo-os da memória. Assim, a memória é constantemente otimizada, e o lixo deixa de ser armazenado.

Se criarmos uma variável $nova recebendo a $conta, ela passará a apontar para a mesma instância de Conta. Nesse ponto, se quisermos remover a referência de $conta, poderemos usar a função unset().



Como ainda temos uma referência à instância criada, ela continuará existindo. Se removermos também a referência em $nova, o coletor de lixo removerá a instância. Esses são os casos em que um objeto/instância deixa de existir. Da mesma forma que temos um código que é executado quando uma instância passa a existir, que é o construtor, existe um código que é executado quando ela deixa de existir, chamado destrutor.

Na classe Conta, podemos criar um método público \_\_destruct() que, se tivermos um $numeroDeContas maior do que 2, exibirá a mensagem "Há mais de uma conta ativa".

public function \_\_destruct()

{

if (self::$numeroDeContas > 2) {

echo "Há mais de uma conta ativa";

}

}COPIAR CÓDIGO

Esse destrutor será executado quando uma das contas deixar de existir. Se executarmos o nosso banco.php, a mensagem será exibida três vezes no console.

Há mais de uma conta ativa3Há mais de uma conta ativaHá mais de uma conta ativa

Isso acontece porque, após criarmos as duas primeiras contas, criamos uma terceira que não é atribuída a nenhuma referência/variável. Assim, o garbage collector imediatamente a destrói. Em seguida, como a execução termina, as outras referências também são destruídas.

Com essa funcionalidade em mente poderíamos, por exemplo, abrir um arquivo no construtor (fopen()) e fechá-lo no destrutor (fclose()). Vamos realizar mais um teste: no destrutor, quando uma referência deixar de existir, decrementaremos o valor de $numeroDeContas em 1.

public function \_\_destruct()

{

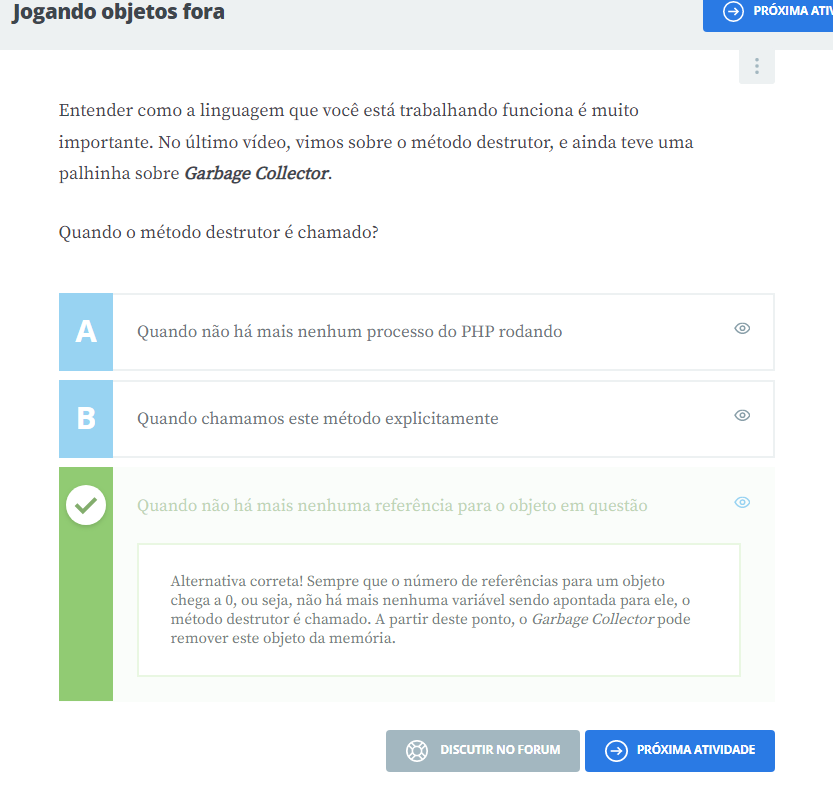
self::$numeroDeContas--;

}COPIAR CÓDIGO

Com isso, quando a instância de Conta que não é atribuída a nenhuma variável deixar de existir, o valor retornado em echo Conta::recuperaNumeroDecontas() será 2. Se salvarmos a instância em uma variável $outra, teremos uma referência para ela no momento da execução, e o valor retornado será 3. Já se fizermos o unset() de $segundaConta, removendo a sua referência, voltaremos a receber 2 contas ativas.

Entendemos quando uma instância deixa de existir na memória, e aprendemos que nesse momento podemos executar o método mágico \_\_destruct(). Se você reparar bem, os métodos que chamamos de métodos mágicos do PHP sempre são precedidos por \_\_ - o \_\_construct, executado quando uma instância é criada, e o \_\_destruct(), quando ela deixa de existir na memória.

Assim, começamos a entender o "ciclo de vida" dos objetos: eles são criados, nós executamos aquilo que é necessário e, em determinado momento, eles são apagados da memória.



Vale ressaltar que é muito importante separar cada classe em seu arquivo, não devemos ter um arquivo com mais de uma classe.

----------------------------------------------------------------------------------------------------

Repare que os dados são exibidos exatamente como antes, mas o var\_dump() de uma conta nos retorna, além do saldo, que por enquanto é um inteiro, um outro objeto do tipo Titular. Este, por sua vez, possui CPF e nome. Ou seja, fizemos com que a classe Conta possua também uma instância de Titular, algo que chamamos de **composição de objetos**.

Um objeto do nosso sistema pode ser composta por vários outros objetos menores - em outras palavras, uma classe pode conter instâncias de outras classes também. Ainda é possível armazenar o titular em outra variável, definir outros métodos na classe Titular, como um método destrutor, entre outras funcionalidades. No nosso caso, essa implementação já está bastante válida, incluindo tudo que já estava funcionando anteriormente, como a validação de um nome.

Agora temos um código mais conciso e compartimentalizado, e é nesse ponto que entra a organização de código, algo que comentamos na primeira aula desse treinamento. Se precisamos de um comportamento relacionado a uma conta, temos a classe Conta para isso; se o comportamento é relacionado ao titular, ele está na classe Titular.

Já entendemos na prática o que é a composição de objetos, e no próximo vídeo conversaremos sobre isso de maneira mais aprofundada.