**Classes e Objetos**

Nesse capítulo começaremos a falar sobre a orientação a objetos, que é o tema deste treinamento. Já vimos como começar a resolver um problema da programação procedural, mas ainda existem vários outros. Por exemplo, no nosso projeto, não temos uma definição exata do que uma conta pode ter, tornando possível, entre outros cenários, a criação de uma $conta somente com o saldo.

$conta = ['saldo' => 5400];

Ainda é possível que a função sacar() funcione nessa conta, pois ela tem um saldo, mas não temos o CPF ou o nome do titular. Ou seja, é uma conta incompleta. A ideia é definirmos uma maneira de criar contas corretamente, algo que estamos fazendo por meio da função criarConta(), que nesse momento não é obrigatória.

Antes de prosseguirmos com o projeto, vamos seguir outro raciocínio. Imagine que você queira fazer um bolo. Isso não é possível simplesmente jogando os ingredientes no forno; é necessário uma forma, que definirá o formato do bolo, seja ele retangular, redondo, etc. A ideia nesse ponto é definirmos uma "forma" para nossa conta.

Deletaremos então o arquivo conta.php e criaremos, no diretório "src", um novo arquivo Conta.php, com o primeiro caractere em caixa alta. Por enquanto isso não fará nenhuma diferença, até porque, pelo menos no Windows, nomes de arquivos não são diferenciados em caracteres minúsculos ou maiúsculos, mas no futuro explicaremos essa distinção.

Queremos fazer com que esse arquivo seja responsável por criar uma conta, e que só seja possível criar uma conta por meio dele, como a "forma" de bolo que citamos anteriormente. No mundo da orientação a objetos, essa "forma" é chamada de **classe**. Sendo assim, usaremos a no nosso código a instrução class, passando em seguida o nome da classe, que será Conta, também com a primeira letra em caixa alta. Por fim, basta abrirmos e fecharmos chaves.

Em PHP, é um padrão, e uma boa prática, criar as classes utilizando a primeira letra em maiúsculo. Se tivéssemos várias palavras, por exemplo "conta-corrente", as palavras subsequentes também se iniciariam com letra maiúscula, como ContaCorrente.

<?php

class Conta

{

// definir dados da conta

}

O escopo da classe, marcado pelas chaves, é onde poderemos definir os dados da conta. A partir de agora temos um novo tipo de dados na nossa aplicação, à semelhança dos tipos string (textos), bool (booleano, que é verdadeiro ou falso), int (inteiros), float (decimais).

Ou seja, poderemos criar variáveis não mais representarão um tipo genérico ou um array, mas especificamente o tipo Conta, e poderemos usá-lo para definir o conteúdo dessas variáveis.

Abriremos então o terminal interativo do PhpStorm (php -a) e faremos require 'src/Conta.php'. Sabemos que para criar uma variável do tipo inteiro, basta atribuirmos a ela um valor inteiro; da mesma forma, se quisermos uma variável do tipo booleano, passamos true ou false, e assim por diante.

$umaNovaConta = 4; //tipo inteiro

$umaNovaConta = false; //tipo booleano

$umaNovaConta = 'Vinicius' //tipo string

Já para criarmos uma variável do tipo Conta, usaremos a instrução new seguida do tipo desejado e um escopo de parênteses (()). Futuramente explicaremos mais a fundo por que esses parênteses existem.

$umaNovaConta = new Conta();COPIAR CÓDIGO

Assim, teremos na variável $umaNovaConta um "endereço" para encontrarmos a Conta que foi criada. Ou seja, não temos nessa variável o valor de tudo que essa Conta tem, mas o endereço onde essas informações estão salvas e por onde conseguiremos acessá-las.

Porém, ainda não temos nenhuma informação nessa Conta, como o CPF e o nome do titular ou o saldo. Aprenderemos a definir esses dados no próximo vídeo.

Antes de prosseguirmos, vamos rever alguns conceitos do vídeo anterior, já que são muito importantes para a temática do treinamento. Nele, conhecemos a palavra-chave class, que nos permite criar uma classe, ou seja, um modelo que nos permitirá criar novos objetos/instâncias, como uma forma de bolo ou a planta de uma casa.

Quando criamos $umaNovaConta usando new Conta(), obtemos como resultado um objeto que não tem todos os valores definidos da conta, mas sim um endereço que aponta para o local onde essa nova conta foi criada. Parece complexo, mas entenderemos melhor esse conceito conforme avançamos.

<?php

class Conta

{

// definir dados da conta

}

Nosso objetivo agora é definirmos os dados de uma conta. Sabemos que uma conta pode ter diversas informações, como agência, endereço da agência, gerente, caixa, número da conta e dígito, código do banco, e assim por diante. Não faz sentido definirmos no nosso sistema tudo que uma conta-corrente tradicional poderia ter se não vamos utilizar tais informações.

Um dos pilares da orientação a objetos é o conceito de **abstração**, que consiste em pegar um conceito real, como uma conta-corrente, e abstrair para o sistema somente o que é necessário para trabalharmos. Até o momento, precisamos do CPF e do nome do titular, além do saldo. Portanto, não estamos interessados no número da conta, na agência, em limite para cheque especial ou outras informações. Em outros sistemas, talvez seja mais interessante armazenar esses dados.

O nome **orientação a objetos** vem do fato de que, a partir de uma classe, podemos criar objetos que servirão de base para montarmos todo o sistema. Sendo assim, o desenvolvimento deixa de ser orientado a funções, como na programação procedural, e se torna orientado a objetos.

Queremos que a nossa conta tenha um $cpfTitular, um $nomeTitular e um $saldo. No PHP, precisamos informar, quando alguém criar um objeto do tipo Conta, se esses dados serão públicos ou privados. No momento queremos que eles sejam públicos, portanto usaremos a palavra-chave public.

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo;

}

Dessa forma, quem tiver uma instância de Conta poderá acessar qualquer uma dessas informações. Feito isso, abriremos novamente o terminal interativo do PhpStorm (php -a) e faremos require 'src/Conta.php'. Em seguida, criaremos uma $primeiraConta passando uma nova instância de Conta, da mesma forma que fizemos no vídeo anterior.

$primeiraConta = new Conta();

Se fizermos um var\_dump() de $primeiraConta, teremos como retorno um objeto do tipo Conta no endereço #1 e contendo os valores ["cpfTitular"], ["nomeTitular"] e ["saldo"].

object(Conta)#1 (3) {

["cpfTitular"]=>

NULL

["nomeTitular"]=>

NULL

["saldo"]=>

NULL

}

Tais "variáveis" que fazem parte de um objeto são chamadas de **atributos**. E o que acontece se definirmos, na nossa classe, que o $saldo da conta é 100?

<?php

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo = 100;

}

Ao fazermos isso, todas as contas criadas terão 100 de saldo, o que não faz sentido nesse caso. Portanto, os valores não devem ser definidos diretamente na classe, mas sim em cada um dos objetos instanciados.

No PHP, para acessarmos um atributo a partir de um objeto usamos uma "seta", representada por ->, como em $primeiraConta->saldo. Em CSS, isso significa o acesso a um ponteiro. Já no PHP, estamos acessando, dentre outras possibilidades, um atributo do objeto de determinado tipo. Podemos, por exemplo, informar $primeiraConta->saldo = 200.

$primeiraConta->saldo = 200;

Se fizermos novamente o var\_dump() de $primeiraConta, o novo saldo será retornado junto às demais informações:

object(Conta)#1 (3) {

["cpfTitular"]=>

NULL

["nomeTitular"]=>

NULL

["saldo"]=>

int(200)

}

Podemos usar a mesma sintaxe para definirmos o $cpfTitular:

$primeiraConta->cpfTitular = '123.456.789-10';

E também para o $nomeTitular:

$primeiraConta->nomeTitular = 'Vinicius Dias';

Com isso, teremos uma conta completamente montada:

object(Conta)#1 (3) {

["cpfTitular"]=>

string(14) "123.456.789-10"

["nomeTitular"]=>

string(13) "Vinicius Dias"

["saldo"]=>

int(200)

}COIAR CÓDIGO

Ao definirmos os atributos da nossa classe, estamos garantindo que toda conta terá um $cpfTitular, um $nomeTitular e um saldo, mesmo que seus valores sejam nulos.

Um detalhe: se no momento desse treinamento você estiver utilizando uma versão a partir do PHP 7.4, você poderá definir também o tipo dos atributos da classe antes mesmo de inicializá-los, como no exemplo:

class Conta

{

public string $cpfTitular;

public string $nomeTitular;

public float $saldo = 100;

}

Assim, é possível garantir que o $cpfTitular gravado sempre será uma string, e que o $saldo sempre será um número. Como no momento da gravação desse curso o PHP 7.4 ainda não estava disponível para Windows, não usaremos as propriedades tipadas.

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo = 100;

}

Nossa classe nos permite a criação de vários objetos, por exemplo uma $segundaConta, recebendo também uma nova instância de Conta, agora vazia.

$segundaConta = new Conta();

Preencheremos essa $segundaConta com os seguintes valores:

$segundaConta->cpfTitular = '987.654.321-10';

$segundaConta->nomeTitular = 'Patricia';

$segundaConta->saldo = 1500;

Se fizermos um var\_dump() de $primeiraConta, ela continuará com os valores originais. Já se fizermos isso para $segundaConta, teremos os valores que acabamos de definir:

object(Conta)#2 (3) {

["cpfTitular"]=>

string(14) "987.654.321-10"

["nomeTitular"]=>

string(8) "Patricia"

["saldo"]=>

int(1500)

}

Ou seja, não definimos os valores na classe, afinal o $saldo não é o mesmo para todos os objetos do tipo Conta. A nossa classe é realmente um modelo que nos permitirá criar instâncias desse tipo, cada uma com os devidos valores para os atributos, que são os da classe. Inclusive, dois objetos de um mesmo tipo podem ter valores diferentes para os seus atributos.

Ainda nessa aula teremos alguns exercícios para fixar esse conteúdo, cuja compreensão é muito importante para a continuidade do treinamento.

Voltando a um ponto já comentado anteriormente, sabemos que na variável $segundaConta não temos todos os valores atribuídos ao objeto, mas sim um endereço que nos permite acessá-los. Justamente por isso, não podemos fazer $segundaConta['saldo'], como faríamos com um array, e só é possível acessar o valor com $segundaConta->saldo.

Conversaremos melhor sobre a diferença entre ter todos os valores e ter o endereço desses valores no próximo vídeo.

**Explicação sobre Orientação à objetos**

Nesse vídeo faremos uma breve revisão do que foi abordado até agora, pois, embora tenhamos escrito pouco código, os conceitos são muito importantes. Primeiro, definimos a estrutura da nossa Conta, como uma forma de bolo ou a planta de uma casa, e dentro dessa definição inicializamos os atributos, ou seja, todos os dados pertencentes às contas que poderemos criar no futuro.

Portanto, sabemos que toda conta criada terá um $cpfTitular, um $nomeTitular e um $saldo. Para deixarmos bem claro o que faz parte de cada uma das instâncias dessa conta, vamos pensar na execução do programa.

Conta

$cpftitular: string

$nomeTitularString

$saldoFloat

Com a nossa Conta pronta, podemos definir uma variável $umaConta recebendo o retorno da expressão new Conta(). Quando esse código é executado, ele separa um espaço para um objeto do tipo Conta, criando o $cpfTitular, o $nomeTitular e o $saldo. Como estamos trabalhando com o PHP 7.3, todos os atributos são inicializados como null.

Se você estiver usando o PHP 7.4 ou superior e tiver definido os tipos dos atributos, eles estarão em um estado não-inicializado.

A variável $umaConta na verdade não contém os valores dos atributos, mas sim o endereço que aponta para o objeto criado. Inclusive, os atributos podem ser acessados ou alterados por meio da variável, usando, por exemplo, $umaConta->saldo.

$umaConta->saldo = 200;

Se criarmos uma variável $segundaConta recebendo também uma nova Conta, o PHP criará um novo espaço de memória com todos os valores nulos (ou sem estarem inicializados). Ou seja, teremos duas variáveis apontando para dois objetos diferentes. Da mesma forma, poderemos acessar os atributos usando $segundaConta->nomeTitular, por exemplo.

$segundaConta->nomeTitular = "Vini";

Mas o que acontece se criarmos uma variável $segundaConta recebendo $umaConta? A primeira variável receberá o valor da segunda, que é o endereço de um objeto do tipo Conta. Como as duas recebem o mesmo endereço, teremos duas referências para o mesmo objeto. Assim, se atribuirmos "200" a $umaConta->saldo e pedirmos a exibição de echo $segundaConta->saldo, o valor retornado será 200.

$umaConta = new Conta();

$umaConta->saldo = 200;

$segundaConta = $umaConta

echo $segundaConta->saldo;

200

Isso acontece pois temos duas variáveis apontando para o mesmo objeto, e essa é a diferença entre referência e para a instância de um objeto em si. Se tivéssemos uma variável "inteira", o valor dela já estaria na variável. Nesse caso, como estamos tratando de uma referência, os valores das variáveis são endereços para chegar até o objeto.

Isso é importante para entendermos como as classes são criadas e referenciadas pelas variáveis, e isso continuará a ser abordado durante o treinamento.

Agora que finalizamos a parte conceitual, vamos colocar nossos conhecimentos em prática, criando códigos que trabalham com variáveis apontando pelo mesmo objeto, entre outros cenários.

**Valor vs Referência**

Estamos trabalhando com a ideia de que uma classe em PHP é uma espécie de "forma de bolo" com a qual podemos criar os nossos objetos. Para fixarmos esse conceito, abriremos o terminal interativo do PhpStorm, faremos novamente o require 'src/Conta.php', criaremos uma $primeiraConta, representando um objeto de Conta, e passaremos as informações necessárias aos seus atributos.

require 'src/Conta.php';

$primeiraConta = new Conta();

$primeiraConta->cpfTitular = '123.456.789-10';

$primeiraConta->nomeTitular = 'Vinicius Dias';

$primeiraConta->saldo = 200;

Como estamos trabalhando no terminal interativo do PhpStorm, as linhas deverão ser executadas uma de cada vez.

Em seguida, criaremos uma $segundaConta na qual definiremos dados diferentes, afinal este é um novo objeto, também do tipo Conta.

$segundaConta = new Conta();

$segundaConta->cpfTitular = '987.654.321-10';

$segundaConta->nomeTitular = 'Patricia';

$segundaConta->saldo = 1500;

Prosseguindo, faremos o var\_dump() da $primeiraConta...

object(Conta)#1 (3) {

["cpfTitular"]=>

string(14) "123.456.789-10"

["nomeTitular"]=>

string(13) "Vinicius Dias"

["saldo"]=>

int(200)

}

...e também da $segundaConta.

object(Conta)#2 (3) {

["cpfTitular"]=>

string(14) "987.654.321-10"

["nomeTitular"]=>

string(8) "Patricia"

["saldo"]=>

int(1500)

}

Tudo isso já havia sido feito nos vídeo anteriores. Se você se lembrar bem, no treinamento anterior a esse aprendemos que, quando passamos argumentos para funções, existe uma diferença entre passá-los como valor e como referência. Sabemos que as variáveis que nos permitem acessar instâncias de Conta são, na verdade, endereços para tais instâncias - ou seja, elas são referências para o endereço de uma conta.

Para que isso fique mais claro, criaremos uma variável $a com o valor 4. Em seguida, criaremos uma variável $b recebendo $a. Como o valor da primeira variável é 4, se fizermos um echo $b, nosso retorno será 4.

$a = 4;

$b = $a;

php > echo $b;

4

Se alterarmos o valor de $b, por exemplo com $b++, a variável assumirá esse novo valor - no caso 5.

php > $b++;

php > echo $b;

5

Porém, o valor de $a não se alterá.

$echo $a;

4

Isso acontece pois a variável $a possui o valor inteiro 4. Porém, se criarmos uma variável $c recebendo $segundaConta e fizermos o seu var\_dump()...

$c = $segundaConta;

var\_dump($c);

...teremos como retorno o mesmo CPF, o mesmo nome do titular e o mesmo saldo.

object(Conta)#2 (3) {

["cpfTitular"]=>

string(14) "987.654.321-10"

["nomeTitular"]=>

string(8) "Patricia"

["saldo"]=>

int(1500)

}

Se você reparar, os endereços são os mesmos, tanto em $c quanto em $segundaConta, e são representados por object(Conta)#2. Portanto, o valor de $c também é o endereço para essa conta. Se executarmos $c->saldo = 2000 para alterarmos o valor do saldo e em seguida fizermos o var\_dump() de $segundaConta, teremos como o retorno uma conta com o saldo modificado.

$c->saldo = 2000;

var\_dump($segundaConta);

object(Conta)#2 (3) {

["cpfTitular"]=>

string(14) "987.654.321-10"

["nomeTitular"]=>

string(8) "Patricia"

["saldo"]=>

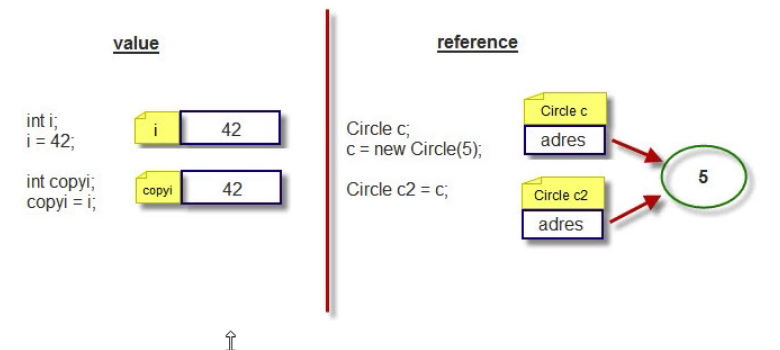
int(2000)

}

Ou seja, a variável $c aponta para o mesmo lugar da variável $segundaConta, já que esta não tem os valores da conta, mas sim o seu endereço.

Imagine, por exemplo, que tenhamos uma variável inteira i com o valor 42. Em seguida, teremos outra variável inteira, copyi, que receberá i. Com isso, teremos na memória dois endereços separados, ainda que com o mesmo valor inteiro.

Quando trabalhamos com classes e objetos, estamos trabalhando com referências. Se criarmos uma instância c da classe Circle e depois criarmos uma nova instância c2 a partir de c, estaremos apontando para o mesmo endereço.



É importante ter isso em mente, pois se passarmos uma instância do tipo Conta por parâmetro para alguma função, por padrão isso será feito por referência, sem que seja necessário adicionar o caractere &, como fizemos no treinamento anterior. Se não tiver consciência disso, é possível que tenhamos comportamentos inesperados no sistema.

Vamos recapitular? Até o momento conseguimos definir um tipo Conta utilizando a palavra-chave class do PHP. Sempre que criarmos uma nova variável desse tipo - ou seja, um objeto ou uma instância de Conta - teremos os atributos que escolhemos, nesse caso $cpfTitular, $nomeTitular e $saldo, e seus valores serão variados de acordo com os objetos criados. Além disso, uma variável que tem o endereço para um objeto do tipo Conta terá com seu valor somente o endereço.

Esses conceitos vão ficando mais claros conforme fazemos exercícios, além de testes usando o terminal interativo do PHP!

Note que ainda temos um problema no nosso código: os comportamentos (ou regras de negócio) do nosso sistema ainda são separados da Conta em si, tornando possível , por exemplo, a operação $primeiraConta->saldo -= 500, que resulta em um valor negativo.

Poderíamos criar uma função saca() que recebe uma Conta, verifica o saldo e possibilita o saque. Entretanto, nada nos obriga a utilizar essa função, sendo possível fazer tais operações diretamente. Sendo assim, como podemos unir os comportamentos de uma conta aos seus dados? Conversaremos sobre isso no próximo capítulo.

**Criando primeiro método**

Já conversamos bastante sobre as diferenças entre classe e objeto/instância, e também sobre as referências para esses objetos. Agora queremos realizar operações com a nossa Conta, que é a classe que criamos, chamando, por exemplo, uma operação sacar() que seja responsável por debitar valores dessa conta, deixando de diminuir o saldo por meio de uma referência.

Nos treinamentos anteriores conhecemos as funções do PHP, e também é possível criarmos funções nas nossas classes. Adicionaremos então a função sacar() recebendo um valorASacar representado pelo tipo float.

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo = 100;

function sacar(float $valorASacar)

{

}

}

Existe ainda um detalhe. Perceba que tivemos que informar se os atributos são públicos ou privados, ou seja, se é possível acessá-los por meio de referências ou não. Por padrão, quando não informamos um modificador de acesso para nossas funções, o PHP assume que elas são públicas.

Entretanto, é sempre recomendado que tenhamos código explícito nos nossos projetos, melhorando a legibilidade, principalmente para pessoas que não estão familiarizadas com o sistema ou com a linguagem. Ou seja, se quisermos manter a função pública, é mais interessante utilizarmos o modificador public.

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo = 100;

public function sacar(float $valorASacar)

{

}

}

Agora que temos a funcionalidade de sacar(), vamos implementá-la. Primeiramente, se $valorASacar for maior do que o $saldo, queremos exibir a mensagem "Saldo indisponível". Do contrário, subtrairemos do $saldo o $valorASacar.

class Conta

{

public $cpfTitular;

public $nomeTitular;

public $saldo = 100;

public function sacar(float $valorASacar)

{

if ($valorASacar > $saldo) {

echo "Saldo indisponível";

} else {

$saldo -= $valorASacar;

}

}

}

Teoricamente esse código está certo, mas o PhpStorm nos informará um erro indicando que a variável $saldo não está definida. Mas por que isso acontece se claramente temos um atributo saldo definido na classe?

Tínhamos uma classe Conta com os seus devidos atributos, e agora adicionamos um comportamento, ao qual damos o nome de **método**. Ou seja, uma função que está dentro de uma classe é chamada de método. Quando chamamos o método sacar(), a classe Conta precisará efetuar uma operação sobre o objeto Conta. Mas como o método saberá sobre qual objeto essa operação será feita, por exemplo quando temos mais de uma conta?

Uma opção seria recebermos uma referência de Conta por parâmetro no método, chamando-a de $contaASacar. Com isso, poderíamos usar $contaASacar->saldo para acessarmos o seu atributo.

public function sacar($contaASacar, float $valorASacar)

{

if ($valorASacar > $contaASacar->saldo) {

echo "Saldo indisponível";

} else {

$contaASacar->saldo -= $valorASacar;

}

}

Isso funciona, pois, ao invés de alterarmos uma variável saldo, alteraremos especificamente esse atributo de uma $contaASacar. Porém, se estivermos chamando o método sacar() a partir de uma variável que aponta para um objeto, será que não temos essa referência de alguma forma dentro do próprio método?

Na verdade temos, e o nome da variável que contém essa referência é $this, uma variável padrão do PHP que se refere à referência atual que chamou o método. Portanto, se $umaConta chamar o método sacar(), o valor de $this será a referência para o objeto representado por $umaConta. Com isso, não precisaremos mais receber o parâmetro $contaASacar.

public function sacar(float $valorASacar)

{

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

echo "Saldo indisponível";

} else {

$this->saldo -= $valorASacar;

}

}

Abriremos o terminal interativo com php -a e incluiremos novamente o arquivo com require 'src/Conta.php'. Definiremos $umaConta cujo saldo será 200, e uma $segundaConta, cujo saldo será 300.

$umaConta = new Conta();

$umaConta->saldo = 200;

$segundaConta = new Conta();

$segundaConta->saldo = 300;

Se fizermos o var\_dump() dessas contas, veremos que elas têm saldos diferentes - ou seja, estamos trabalhando com objetos diferentes de uma mesma classe.

Ainda no terminal interativo, testaremos nosso novo método com $umaConta->sacar(), passando o valor 150. Sabemos que os atributos são específicos de cada objeto, mas os métodos não precisam ser copiados de uma conta para outra, valendo para todas elas. Ou seja, quando chamamos sacar(), o PHP acessa a definição da classe para executar esse método, encontrando também a variável $this, que faz referência ao objeto que chamou o método.

Após a execução, se fizermos o var\_dump() de $umaConta, teremos um novo saldo no valor de 50.

object(Conta)#1 (3) {

["cpfTitular"]=>

NULL

["nomeTitular"]=>

NULL

["saldo"]=>

float(50)

}

Entretanto, a $segundaConta continuará com o valor original de 300, sem nenhuma alteração. Dessa vez, executaremos $segundaConta->sacar(30). Como resultado, teremos um saldo de 270, algo que podemos conferir com o var\_dump().

object(Conta)#2 (3) {

["cpfTitular"]=>

NULL

["nomeTitular"]=>

NULL

["saldo"]=>

float(270)

}

Vamos recapitular? Na classe Conta temos um método sacar() que não é copiado para cada objeto/instância, mas dentro dele temos uma referência para o objeto que o chamou. Com isso, conseguimos saber quais são os atributos do objeto que estamos alterando, sendo possível modificar, por exemplo, o atributo saldo da conta que chamou o método.

A utilização da variável $this pode ser um conceito confuso, mas é muito importante entendê-la para se aprofundar na orientação a objetos com o PHP. Para praticarmos, no próximo vídeo implementaremos a funcionalidade de depositar.

**Métodos com referência**

Nosso objetivo agora é desenvolver um método que faz transferências entre uma conta e outra. No curso anterior, tínhamos um método $titularComLetrasMaiusculas() no qual recebíamos uma referência como parâmetro, no caso a conta original ($conta).

function titularComLetrasMaiusculas(array &$conta)

{

$conta['titular'] = mb\_strtoupper($conta['titular']);

}

O procedimento agora será semelhante, recebendo uma conta para a qual transferiremos um valor, modificando o seu saldo. Utilizando o PhpStorm, digitar pubf seguido de "Enter" (ou "Tab") já monta automaticamente a estrutura de um método.

public function ()

{

}

Chamaremos esse método de transferir(), passaremos como parâmetros um $valorATransferir, do tipo float, e uma $contaDestino, um objeto do tipo Conta. Esse método também não terá um retorno, portanto usaremos a instrução void.

public function transferir(float $valorATransferir, Conta $contaDestino): void

{

}

Verificaremos então se o $valorAtransferir é maior do que o saldo da conta de origem. Em caso positivo, exibiremos a mensagem de erro "Saldo indisponível". Do contrário, realizaremos a transferência.

Para isso, realizaremos uma chamada de sacar() a partir da conta atual ($this), passando como parâmetro o $valorATransferir. Em seguida, chamaremos depositar() a partir da $contaDestino, passando também o $valorATransferir.

public function transferir(float $valorATransferir, Conta $contaDestino): void

{

if ($valorATransferir > $this->saldo) {

echo "Saldo indisponível";

} else {

$this->sacar($valorATransferir);

$contaDestino->depositar($valorATransferir);

}

}

Para testarmos, abriremos o terminal interativo do PHP e inicializaremos as variáveis $contaUm e $contaDois, ambas recebendo instâncias de Conta. Em seguida, faremos um depósito com o valor 500 em $contaUm, mantendo a $contaDois com seu valor inicial (0).

$contaUm = new Conta();

$contaDois = new Conta();

$contaUm->depositar(500);

Chamaremos então, a partir de $contaUm, o método transferir(), passando como parâmetros o valor 200 e a $contaDois. Se fizermos um echo de $contaUm->saldo, veremos que o valor da transferência foi debitado com sucesso.

$contaUm->transferir(200, $contaDois);

echo $contaUm->saldo;COPAR CÓDIGO

300

Da mesma forma, um echo $contaDois->saldo nos mostrará que o valor foi transferido corretamente para a outra conta.

echo $contaDois->saldo;

200

Repare que passamos a $contaDois por parâmetro e conseguimos modificá-la dentro do método transferir(), mostrando que essas variáveis são referências, e não valores diretos.

Nos nossos métodos, estamos sempre verificando possíveis erros antes de continuar a execução. Porém, existe uma regra na programação - que não vamos entrar em detalhes sobre a qual linha de pensamento ela pertence - afirmando que devemos evitar ao máximo a utilização de else. Será que isso é aplicável ao nosso caso?

Como o método sacar() não devolve nada (void), podemos adicionar a instrução return ao corpo do if, fazendo com que a execução pare ao chegar nessa linha. Com isso, poderemos simplesmente remover o else.

public function sacar(float $valorASacar): void

{

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

echo "Saldo indisponível";

return;

}

$this->saldo -= $valorASacar;

}

Assim, se o $valorASacar for maior do que $this->saldo, a mensagem definida será exibida e a execução do método parará. Portanto, tudo que vier após o if só será executado se a nossa verificação der certo, o que também remove um nível de identação e ajuda na legibilidade. Repetiremos esse processo para os outros métodos da classe Conta.

public function depositar(float $valorADepositar): void

{

if ($valorADepositar < 0) {

echo "Valor precisa ser positivo";

return;

}

$this->saldo += $valorADepositar;

}

public function transferir(float $valorATransferir, Conta $contaDestino): void

{

if ($valorATransferir > $this->saldo) {

echo "Saldo indisponível";

return;

}

$this->sacar($valorATransferir);

$contaDestino->depositar($valorATransferir);

}

Assim temos um código um pouco mais sucinto. Porém, se você prefere ou entende melhor o código com o else, não é um problema, só estamos conhecendo outras possibilidades.

Prosseguindo, repare que se executarmos $contaDois=>saldo -= 500 no terminal, não receberemos erro algum, resultando em um saldo negativo (-300). Ou seja, já temos todo o comportamento da conta junta aos dados, mas nenhuma forma de prevenir o acesso a esses dados de forma direta. Conversaremos mais sobre como proteger a nossa conta no próximo capítulo.

**Impedindo o acesso**

Já sabemos definir uma classe e criar um objeto a partir dela, e conhecemos os atributos e métodos, além de como trabalhar com eles. Também vimos algumas boas práticas de programação na orientação a objetos, como o *early return*, que retorna um método logo após a ocorrência de um erro, finalizando a sua execução.

Continuaremos a implementar melhorias no nosso sistema, a começar pelo problema que identificamos ainda no começo do treinamento: é possível modificar o saldo de uma conta livremente, por exemplo por meio do terminal interativo do PHP. Aliás, de modo a reduzirmos a utilização desse terminal no nosso desenvolvimento, criaremos na raiz do projeto um arquivo banco.php, no qual passaremos a realizar as nossas operações.

Nele, importaremos o arquivo de conta com require\_once 'src/Conta.php' e criaremos uma instância de Conta chamada $primeiraConta.

<?php

require\_once 'src/Conta.php';

$primeiraConta = new Conta();

Perceba que a IDE nos ajuda a escrever os códigos, disponibilizando diversas opções de autopreenchimento após $primeiraConta->, como depositar(), sacar(), saldo e assim por diante. Além disso, a letra ao lado da opção nos ajuda a identificar a que ela se refere, como **m*\* de \*method* (método) e \*\*f** de *field* ("campo").

A ideia é impedirmos que operações com o saldo sejam feitas diretamente, sem usar os métodos correspondentes, como em $primeiraConta->saldo -=300. Além de ser pouco seguro, essa operação resultaria em um saldo negativo, algo que o método sacar() impede que aconteça com a verificação que implementamos na sua execução.

<?php

require\_once 'src/Conta.php';

$primeiraConta = new Conta();

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

$primeiraConta->saldo -=300; // isso não é ok

A ideia é tornarmos o campo $saldo privado, de forma que somente a própria Conta consiga acessá-lo. Ou seja, ao criarmos a $primeiraConta, somente essa instância terá acesso ao seu atributo por meio dos métodos que ela pode executar.

class Conta

{

public $cpfTitular;

public $nomeTitular;

private $saldo = 0;

Agora, se tentarmos executar o arquivo banco.php, o que pode ser feito clicando com o botão direito e em seguida em "Run > banco.php (PHP Script)", receberemos um erro:

Saldo indisponível

Fatal error: Uncaught Error: Cannot access private property Conta::$saldo in C:\Users\Rodrigo\Desktop\php-oo\projeto\banco.php:7

Note que recebemos a mensagem de erro do método sacar(), já que 300 é um valor maior do que o saldo da conta (que por padrão definimos como 0), e depois um erro do próprio PHP informando que não é possível acessar uma propriedade privada.

Com isso, conseguimos exatamente o que planejávamos. Saiba que é uma recomendação geral na orientação a objetos que tenhamos apenas propriedades privadas nas classes, e que somente os métodos sejam públicos. Faremos essas alterações em nosso código:

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo = 0;

public function sacar(float $valorASacar): void

{

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

echo "Saldo indisponível";

return;

}

$this->saldo -= $valorASacar;

}

public function depositar(float $valorADepositar): void

{

if ($valorADepositar < 0) {

echo "Valor precisa ser positivo";

return;

}

$this->saldo += $valorADepositar;

}

public function transferir(float $valorATransferir, Conta $contaDestino): void

{

if ($valorATransferir > $this->saldo) {

echo "Saldo indisponível";

return;

}

$this->sacar($valorATransferir);

$contaDestino->depositar($valorATransferir);

}

}

De volta ao arquivo banco.php, removeremos a linha que fazia um saque diretamente e adicionaremos, antes da chamada de sacar(), um depósito com o valor 500.

require\_once 'src/Conta.php';

$primeiraConta = new Conta();

$primeiraConta->depositar(500);

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

Ao executarmos o código novamente, não teremos nenhum erro. Porém, como definimos nossas propriedades como privadas, não conseguiremos mais exibir o saldo ou definir o nome titular e o CPF. Conversaremos sobre como resolver esse problema no próximo vídeo.

**Encapsulamento**

No [exercício anterior](https://cursos.alura.com.br/course/php-oo-classes-metodos-atributos/task/67348), deixamos um link para uma postagem no Blog da Caelum falando sobre como não aprender **getters and setters**, que são justamente os métodos de acesso que acabamos de criar.

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. Porém, se não existe nenhuma validação referente a aquele atributo, não existe problema em mantê-lo público, ainda que sejam raros os casos em que isso faz sentido.

Isso também pode ser definido pensando no futuro. No nosso projeto, é esperado que um dia tenhamos uma validação para definir o CPF e o nome do titular. Como é uma classe que pode ter regras futuramente, já podemos ter os métodos de acesso.

A ideia de não termos mais acesso direto à propriedade $saldo, fazendo com que só seja possível sacar, depositar ou transferir para modificá-la, é um dos pilares da orientação a objetos, e se chama **encapsulamento**.

O encapsulamento consiste na necessidade de proteger as instâncias da nossa classe do mundo externo e do próprio usuário, garantindo que ele não viole as nossas regras de negócio. Além disso, também devemos facilitar a utilização da classe por meio dos seus métodos.

Por exemplo, se um dia implementarmos uma taxa de saque, o usuário ou mesmo o desenvolvedor que estiver usando o método saca() não precisa escrevê-la manualmente, já que ela pode ser aplicada diretamente na definição do método.

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

Anteriormente foi comentado que existe outra forma de inicializarmos um atributo. Além disso, quando criamos uma instância de Conta(), ela não possui um titular ou um CPF, e não faz sentido termos uma conta sem esses valores.

Será possível executarmos algum código nesse momento, obrigando que o usuário passe algumas informação para que a criação de uma conta seja efetuada com sucesso? Conversaremos sobre isso no próximo capítulo.

**Método Construtor**

No vídeo anterior comentamos sobre inicializar os valores dos atributos na própria classe. Antes de prosseguirmos, faremos um var\_dump() logo após a criação de $primeiraConta para entendermos se ela está em um estado consistente, ou seja, se ela faz sentido para as nossas regras de negócio. Por exemplo, para a nossa aplicação, é esperado que o $saldo sempre esteja zerado na criação de uma nova conta.

object(Conta)#1 (3) {

["cpfTitular":"Conta":private]=>

NULL

["nomeTitular":"Conta":private]=>

NULL

["saldo":"Conta":private]=>

int(0)

}

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;

}

//...

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 como no exemplo abaixo.

require\_once 'src/Conta.php';

$primeiraConta = new Conta();

$primeiraConta = new Conta();

$primeiraConta = new Conta();

$primeiraConta = new Conta();

var\_dump($primeiraConta);

$primeiraConta->deposita(500);

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

$primeiraConta->defineCpfTitular('123.456.789-10');

echo $primeiraConta->recuperaSaldo();

echo $primeiraConta->recuperaCpfTitular();

Criando uma nova conta

Criando uma nova conta

Criando uma nova conta

Criando uma nova conta

Feito esse teste, removeremos as contas duplicadas e o var\_dump() de banco.php. Agora que temos um ponto onde um código sempre é executado, podemos, por exemplo, passar a definir o $saldo no construtor.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

public function \_\_construct()

{

$this->saldo = 0;

}

//...

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;

}

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().

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->recuperaSaldo();

echo $primeiraConta->recuperaCpfTitular();

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;

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();

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');

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.

Para testarmos, faremos um var\_dump() da nova conta.

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

var\_dump($segundaConta);

Assim, teremos um novo objeto do tipo Conta, este com o saldo zerado já que ainda não fizemos nenhum depósito.

object(Conta)#2 (3) {

["cpfTitular":"Conta":private]=>

string(14) "698.549.548-10"

["nomeTitular":"Conta":private]=>

string(8) "Patricia"

["saldo":"Conta":private]=>

int(0)

}

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.

**Método Privado**

Nosso objetivo agora é garantir que todas as instâncias de Conta estejam em um estado válido no momento da sua criação. Já é obrigatório passar um CPF e um nome do titular. Como a validação de um CPF é complexa, não a discutiremos nesse treinamento, de modo a mantermos o foco na orientação a objeto. Sendo assim, trabalharemos com o nome do titular.

Para que o $nomeTitular seja válido, definiremos que ele precisa ter pelo menos 5 caracteres. Portanto, uma usuária com o nome Ana precisará fornecer também o sobrenome.

Uma forma de implementarmos essa validação é incluindo, no construtor, um operador if que verifica, com a função strlen(), se o tamanho da string $nomeTitular é menor que 5. Em caso afirmativo, mostraremos a mensagem de erro "Nome precisa ter pelo menos 5 caracteres" e pararemos a execução do programa com o exit().

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

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

$this->saldo = 0;

}

Existem formas melhores de tratar erros de modo a passarmos um feedback mais interessante ao desenvolvedor que está chamando esse método, mas abordaremos esse assunto em treinamentos específicos sobre exceções (*exceptions*). Em banco.php, alteraremos o nome do titular de $segundaConta para Ana.

$segundaConta = new Conta('698.549.548-10', 'Ana');

var\_dump($segundaConta);

Executando o código, como planejado, a mensagem "Nome precisa ter pelo menos 5 caracteres" será exibida no console. Porém, imagine que tenhamos também a validação do CPF, que é mais complexa. Manter todo esse código no construtor provavelmente traria problemas de legibilidade.

É interessante que, durante o desenvolvimento, cada método tenha apenas uma responsabilidade. Nesse caso, o construtor está inicializando as variáveis e validando-as. Sendo assim, separaremos a validação do atributo $nomeTitular em outro método, que chamaremos de validaNomeTitular().

O novo método receberá uma string $nomeTitular por parâmetro e simplesmente executará a mesma verificação que criamos anteriormente.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

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

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

$this->validaNomeTitular($nomeTitular);

$this->saldo = 0;

}

//... código omitido ...

public function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

}

Em programação, tal processo é chamado de **refatoração**, e consiste em melhorar o código sem alterar os comportamentos. Tanto é que, se executarmos novamente banco.php, teremos exatamente a mesma resposta.

Entretanto, repare que agora é possível chamar $primeiraConta->validaNomeTitular(), passando como parâmetro um nome qualquer, mesmo que ele não esteja associado a uma conta. Como discutimos anteriormente ao estudarmos o encapsulamento, só devemos expôr aquilo que é necessário da nossa aplicação, e o método validaNomeTitular() é uma regra interna que somente a classe Conta precisa acessar. Sendo assim, transformaremos tal método em privado (private).

private function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

Via de regra, os atributos devem ser privados e somente os métodos públicos. Entretanto, nem todos os métodos devem ser públicos. Quando temos códigos que só são executados dentro da classe, é perfeitamente aceitável que eles sejam privados. Assim, o validaNomeTitular() só será acessível na instância de Conta, e poderá ser chamado no construtor, mas não em outros arquivos.

É interessante termos métodos privados para organizarmos o código da classe, mantendo a legibilidade sem expor os comportamentos internos da aplicação. Além disso, faz mais sentido termos métodos mais objetos, com uma única responsabilidade, o que às vezes traz a necessidade de extrairmos outros métodos.

No próximo vídeo conversaremos sobre como saber quantas contas foram criadas durante a execução do nosso programa.

**Métodos Estáticos**

Nosso objetivo é encontrarmos uma forma de armazenar o número de contas criadas. Se a ideia é fazer uma verificação no momento da criação de uma conta, podemos utilizar o construtor da classe. Sendo assim, criaremos um atributo público $numeroDeContas que será inicializado com o valor 0 e, no construtor, será incrementado sempre que uma instância de Conta for criada.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

public $numeroDeContas = 0;

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

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

$this->validaNomeTitular($nomeTitular);

$this->saldo = 0;

$this->numeroDeContas++;

}

//...

Vamos refletir sobre esse código. De que onde exatamente queremos exibir o $numeroDeContas? De $primeiraConta ou de $segundaConta? Na verdade, nenhuma dessas possibilidades faz sentido, afinal não queremos ter um atributo para cada conta.

Quando criarmos a $primeiraConta, o $numeroDeContas será 0 e o construtor adicionará 1 à contagem. Da mesma forma, quando criarmos a $segundaConta, também faremos a incrementação a partir do 0. Ou seja, o $numeroDeContas sempre será 1 para cada instância de Conta.

O que queremos não é um atributo de cada instância/objeto, mas sim um atributo geral da própria classe. Para isso, é possível criarmos um atributo estático (static).

Também é possível criar métodos estáticos!

Esse tipo de atributo não pode ser acessado por meio de $this, afinal essa variável faz referência à instância da classe, mas sim por meio de do próprio nome da classe - nesse caso, Conta::$umeroDeContas.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

public static $numeroDeContas = 0;

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

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

$this->validaNomeTitular($nomeTitular);

$this->saldo = 0;

Conta::$numeroDeContas++;

}

//...

Com isso, temos um atributo estático da classe Conta que é inicializado como 0 e incrementado cada vez que uma instância é criada. em banco.php, após a instanciação das nossas contas, faremos um echo de Conta::$numeroDeContas.

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;

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

var\_dump($segundaConta);

echo Conta::$numeroDeContas;

Como retorno, junto das outras informações que imprimimos no console, teremos o número 2, que é o total de contas criadas. Se incluirmos mais uma chamada de new Conta(), mesmo que não salvemos essa instância em uma variável, teremos um total de 3 contas.

<?php

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;

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

var\_dump($segundaConta);

new Conta('123', 'Abcdefg');

echo Conta::$numeroDeContas;

3

Como comentamos anteriormente, na maioria das vezes é interessante mantermos os atributos privados e os métodos públicos. Do contrário, poderíamos, por exemplo, atribuir um número qualquer a Conta::$numeroDeContas.

Conta::$numeroDeContas - 1234;COPIAR CÓDIGO

Sendo assim, manteremos o atributo $numeroDeContas como private, já que o construtor da classe conseguirá acessá-lo normalmente. Entretanto, não conseguiremos mais acessar o seu valor de fora da classe - ou seja, precisaremos de um método de acesso estático para isso. Esse método, que chamaremos de recuperaNumeroDeContas(), devolvará um inteiro Conta::$numeroDeContas.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

private static $numeroDeContas = 0;

//... código omitido

public static function recuperaNumeroDeContas(): int

{

return Conta::$numeroDeContas;

}

}

Feito isso, em banco.php, poderemos chamar Conta::recuperaNumeroDeContas() para obtermos esse valor.

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;

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

var\_dump($segundaConta);

new Conta('123', 'Abcdefg');

echo Conta::recuperaNumeroDeContas();

3

Assim, chamaremos um método estático que, por sua vez, acessará um atributo estático que é incrementado toda vez que uma nova instância é criada.

Aprendemos o conceito de atributos e métodos estáticos, e vimos que é possível mantê-los privados sem nenhum problema. Quando acessamos uma instância, fazemos isso por meio do nome reservado $this. Existe também um nome reservado para acessarmos a classe atual, que é self. Ele é útil, por exemplo, quando o nome da nossa classe é mais extenso e cheio de especificações, como ContaCorrenteQueNaoPossuiTarifa. Além disso, se porventura renomearmos a classe, não precisaremos renomear também todas as chamadas.

class Conta

{

private $cpfTitular;

private $nomeTitular;

private $saldo;

private static $numeroDeContas = 0;

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

{

$this->cpfTitular = $cpfTitular;

$this->nomeTitular = $nomeTitular;

$this->validaNomeTitular($nomeTitular);

$this->saldo = 0;

self::$numeroDeContas++;

}

//... código omitido...

public static function recuperaNumeroDeContas(): int

{

return self::$numeroDeContas;

}

}

Feita essa alteração, nosso código continuará funcionando normalmente, sem nenhuma mudança no comportamento. Existem ainda alguns outros detalhes - por exemplo, no retorno da função recuperaNumeroDeContas(), poderíamos substituir o self por static, e o código funcionaria da mesma maneira. Entretanto, não entraremos em detalhes sobre isso, pois é assunto de vídeos futuros.

Recapitulando: nossa classe Conta possuía atributos de cada uma das instâncias criadas a partir dela, e agora temos também um atributo $numeroDeContas que se refere à classe em si. Nossa classe poderia ter outros atributos estáticos, que não precisariam estar presentes em cada uma das instâncias - por exemplo, um $codigoBanco, já que todas as contas criadas a partir dessa classe fazem parte do mesmo banco.

Ainda falta um detalhe para finalizarmos este capítulo. Estamos inicializando os atributos da classe no construtor. Imagine que tenhamos uma conexão com um banco de dados, ou que tenhamos salvado informações em um arquivo, e queremos desfazer essas alterações. Será que existe um código que é executado quando a instância deixa de existir, por exemplo quando a execução do programa acaba ou quando excluímos uma variável com o unset()? Conversaremos sobre isso no próximo vídeo.

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

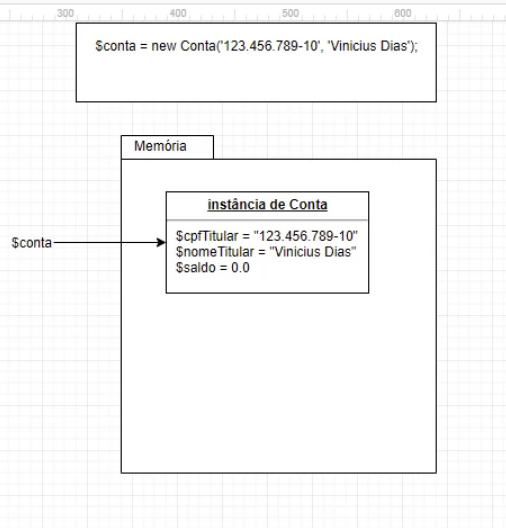
Apesar de parecerem úteis, podem ser bastante danosos, se escritos indiscriminadamente.

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

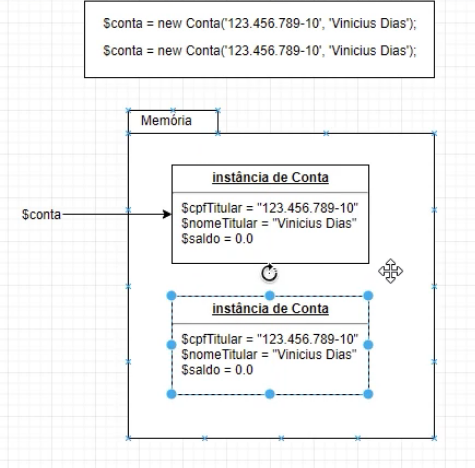
**Apenas escreva métodos estáticos quando eles realmente forem necessários e fizerem sentido (é raro).**

**Método Destrutor (Garbage Collector)**

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.



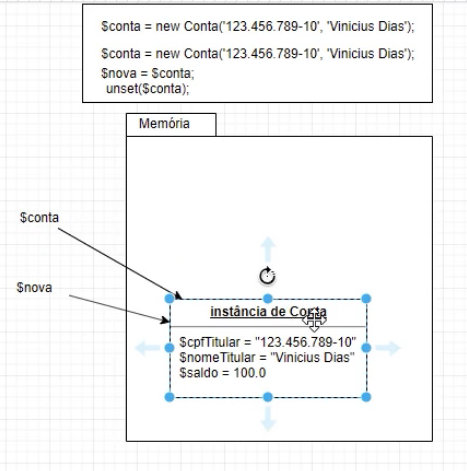
Se criarmos uma nova $conta, com os mesmos dados ou não, teremos uma nova instância.



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";

}

}

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--;

}

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.

Já conseguimos criar tipos complexos, com diversas propriedades. Entretanto, se temos um $cpfTitular e um $nomeTitular, não seria mais interessante termos uma classe que representa esse titular? Conversaremos sobre isso no próximo capítulo.

**Classe Titular**

Chegamos ao capítulo final do nosso treinamento de introdução à orientação a objetos utilizando PHP. Ainda existem alguns pontos que podem ser melhorados na nossa classe, como a definição dos atributos $cpfTitular e $nomeTitular em uma classe Conta, além da validação deste último, que também não parece condizer com a responsabilidade dessa classe.

Quando temos muitos atributos não tão relacionados à classe em si, mas relacionados entre si, parece ser mais interessante criar outra classe para representar esses atributos. Além disso, a existência de um comportamento que envolve esses atributos é um indício ainda maior de que eles devem ser extraídos para uma classe separada.

Segundo nosso modelo, uma conta possui saldo e titular, titular este que possui CPF e nome. Como não é a conta em si que possui CPF e nome, faz sentido modelarmos uma entidade Titular, refinando a abstração do nosso projeto.

Clicaremos com o botão direito na pasta "src" e em seguida em "New > PHP Class" de modo a criarmos a nova classe. Nela, definiremos os atributos privados $cpf e $nome e criaremos um construtor \_\_construct(), algo que pode ser facilitado digitando apenas \_\_cons e apertando "Tab" ou "Enter".

Nosso construtor receberá por parâmetro as strings $cpf e $nome, e faremos as suas atribuições no corpo do método. Esse processo também pode ser facilitado no PhpStorm usando o atalho "Alt + Enter" e a opção "*Initialize Fields*" (ou "*Initialize Properties*").

class Titular

{

private $cpf;

private $nome;

public function \_\_construct(string $cpf, string $nome)

{

$this->cpf = $cpf;

$this->nome = $nome;

}

}

Outro atalho interessante do PhpStorm é o "Alt + Insert", que nos dá acesso à opção "Getters" e nos permite gerar automaticamente os métodos de acesso dos atributos da classe. Feito isso, removeremos os comentários desnecessários e renomearemos os métodos para recuperaCpf() e recuperaNome(), respectivamente.

class Titular

{

private $cpf;

private $nome;

public function \_\_construct(string $cpf, string $nome)

{

$this->cpf = $cpf;

$this->nome = $nome;

}

public function recuperaCpf(): string

{

return $this->cpf;

}

public function recuperaNome(): string

{

return $this->nome;

}

}

Com isso, não teremos mais na classe Conta os atributos $cpfTitular e $nomeTitular, passando a ter somente um $titular que receberemos por parâmetro no construtor.

class Conta

{

private $titular;

private $saldo;

private static $numeroDeContas = 0;

public function \_\_construct(Titular $titular)

{

$this->titular = $titular.

$this->saldo = 0;

self::$numeroDeContas++;

}

//...

Além disso, removeremos os métodos recuperaCpfTitular() e recuperaNomeTitular(), que não têm mais uso, e moveremos o método validaNomeTitular() para a classe Titular (algo que pode ser feito simplesmente recortando-o e colando-o). Passaremos, então, a chamar a validação dentro do construtor dessa classe, logo antes de definirmos o $nome.

class Titular

{

private $cpf;

private $nome;

public function \_\_construct(string $cpf, string $nome)

{

$this->cpf = $cpf;

$this->validaNomeTitular($nome);

$this->nome = $nome;

}

private function validaNomeTitular(string $nomeTitular)

{

if (strlen($nomeTitular) < 5) {

echo "Nome precisa ter pelo menos 5 caracteres";

exit();

}

}

//...

Agora a classe Conta só possui informações relacionadas a uma conta, e conseguimos sacar, depositar, transferir e recuperar o saldo. Já os atributos e ações referentes a um titular estão armazenados na classe Titular.

Corrigiremos então alguns pontos do nosso arquivo banco.php, começando pela instanciação de uma nova conta, que agora precisará também da criação de um objeto new Titular(). Além disso, a classe Titular está em outro arquivo - o que é bastante recomendado na programação orientada a objetos - e portanto deveremos buscá-lo com `require\_once 'src/Titular.php'.

require\_once 'src/Conta.php';

require\_once 'src/Titular.php';

$primeiraConta = new Conta(new Titular('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;

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

var\_dump($segundaConta);

$outra = new Conta(new Titular('123', 'Abcdefg'));

echo Conta::recuperaNumeroDeContas();

Ainda temos um problema: os métodos recuperaNomeTitular() e recuperaCpfTitular() não existem mais. Nesse caso, as alternativas são implementarmos novamente esses métodos, agora chamando o recuperaCpf() e recuperaNome() da classe Titular, ou criarmos um método recuperaTitular() que devolve um objeto do tipo Titular a partir do qual poderíamos chamar os métodos citados.

Faremos a implementação da primeira forma, implementando novamente na classe Conta os métodos recuperaNomeTitular() e recuperaCpfTitular(). Ambos retornarão strings conseguidas a partir da chamada de $this->titular->recuperaNome() e $this->titular->recuperaCpf(), respectivamente.

public function recuperaNomeTitular(): string

{

return $this->titular->recuperaNome();

}

public function recuperaCpfTitular(): string

{

return $this->titular->recuperaCpf();

}

Com isso nosso código voltará a funcionar corretamente, e nenhum erro será gerado na execução de banco.php. Como retorno no console, teremos uma estrutura um pouco diferente:

Vinicius Dias

123.456.789-10

200

object(Conta)#3 (2) {

["titular":"Conta":private]=>

object(Titular)#4 (2) {

["cpf":"Titular":private]=>

string(14) "698.549.548-10"

["nome":"Titular":private]=>

string(8) "Patricia"

}

["saldo":"Conta":private]=>

int(0)

}

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.

**Criando Classe jeito prático**

Nesse vídeo iremos relembrar o conteúdo que aprendemos no treinamento anterior. Portanto, se você estiver bem confiante e fez o curso recentemente, poderá seguir alguns passos e então pular para o próximo vídeo. A ideia é adicionarmos um novo atributo à classe Titular para representar o seu endereço, representado por uma nova classe.

Começaremos criando, em "src", a classe Endereco. Nela teremos os atributos privados $cidade, $bairro, $rua e $numero. Como dito no curso anterior, se estiver usando o PHP 7.4, você poderá definir os tipos desses atributos como no exemplo abaixo.

class Endereco

{

private string $cidade;

private string $bairro;

private string $rua;

private string $numero;

}

Note que definimos o $numero como uma string, e não como um inteiro. Isso porque, se trabalharmos com o $numero sendo um inteiro, podemos ter desperdícios de memória (se utilizarmos números muito grandes) ou termos uma representação não muito consistente. Por exemplo, não conseguiríamos representar "71b" com um inteiro. A ideia central é: se não realizaremos cálculos com os valores de alguma propriedade ou variável, não precisamos defini-la como um tipo numérico.

Como nesse treinamento estamos utilizando o PHP 7.3, versão na qual a tipagem de atributos ainda não era possível, removeremos essas definições do nosos código. No PhpStorm, podemos usar o atalho "Alt + J" do teclado para selecionar vários textos iguais, o que facilitará essa edição.

class Endereco

{

private $cidade;

private $bairro;

private $rua;

private $numero;

}

Prosseguindo, criaremos um construtor \_\_construct() para inicializar todos os atributos. No PhpStorm, o atalho "Alt + Insert" nos dá acesso à opção "Constructor...", que gera automaticamente um construtor com os atributos selecionados na janela que é aberta em seguida. No nosso caso, selecionaremos todos eles. Depois disso, removeremos o comentário gerado pela IDE e definiremos todos os atributos recebidos no construtor como strings.

class Endereco

{

private $cidade;

private $bairro;

private $rua;

private $numero;

public function \_\_construct(string $cidade, string $bairro, string $rua, string $numero)

{

$this->cidade = $cidade;

$this->bairro = $bairro;

$this->rua = $rua;

$this->numero = $numero;

}

}

Além do construtor, precisaremos dos métodos de acesso, ou seja, dos nossos *getters*, que também criaremos por meio do atalho "Alt + Insert > Getters...".

class Endereco

{

private $cidade;

private $bairro;

private $rua;

private $numero;

public function \_\_construct(string $cidade, string $bairro, string $rua, string $numero)

{

$this->cidade = $cidade;

$this->bairro = $bairro;

$this->rua = $rua;

$this->numero = $numero;

}

public function getCidade(): string

{

return $this->cidade;

}

public function getBairro(): string

{

return $this->bairro;

}

public function getRua(): string

{

return $this->rua;

}

public function getNumero(): string

{

return $this->numero;

}

}

Como temos feito nossos códigos em português, não faz sentido usarmos o verbo get. Portanto, renomearemos nossos métodos com o verbo recupera().

public function recuperaCidade(): string

{

return $this->cidade;

}

public function recuperaBairro(): string

{

return $this->bairro;

}

public function recuperaRua(): string

{

return $this->rua;

}

public function recuperaNumero(): string

{

return $this->numero;

}

Com o endereço criado, voltaremos à classe Titular e adicionaremos ao construtor o parâmetro Endereco $endereco. Com "Alt + Enter > Initialize Fields" (ou "Initialize Properties"), o PhpStorm se encarregará de adicionar a nova propriedade e fazer a sua atribuição no próprio construtor.

class Titular

{

private $cpf;

private $nome;

private $endereco;

public function \_\_construct(CPF $cpf, string $nome, Endereco $endereco)

{

$this->cpf = $cpf;

$this->validaNomeTitular($nome);

$this->nome = $nome;

$this->endereco = $endereco;

}

//...

Ainda nessa classe, criaremos também o método recuperaEndereco() que simplesmente retorna $this->endereco. Em banco.php, os pontos em que criamos uma nova conta passarão a apresentar um erro, afinal, além do CPF e do nome, precisamos passar também um endereço.

No nosso caso, imaginaremos que os usuários "Vinicius Dias" e "Patricia" são casados e moram no mesmo endereço. Criaremos então uma variável $endereco recebendo uma nova instância de Endereco com a cidade "Petrópolis", o bairro "um bairro", a rua "minha rua" e o número "71B".

Temos também uma terceira conta, armazenada na variável $outra, para qual criaremos um $outroEndereco com os dados 'A', 'b', 'c' e '1D' representando, respectivamente, cidade, bairro, rua e número.

require\_once 'src/Conta.php';

require\_once 'src/Titular.php';

require\_once 'src/CPF.php';

$endereco = new Endereco('Petrópolis', 'um bairro', 'minha rua', '71B');

$vinicius = new Titular(new CPF('123.456.789-10'), 'Vinicius Dias', $endereco);

$primeiraConta = new Conta($vinicius);

$primeiraConta->deposita(500);

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

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

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

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

$patricia = new Titular(new CPF('698.549.548-10'), 'Patricia', $endereco);

$segundaConta = new Conta($patricia);

var\_dump($segundaConta);

$outroEndereco = new Endereco('A', 'b', 'c', '1D');

$outra = new Conta(new Titular(new CPF('123.654.789-01'), 'Abcdefg', $outroEndereco));

unset($segundaConta);

echo Conta::recuperaNumeroDeContas();

Se executarmos o código dessa forma, teremos um erro, já que nos esquecemos de fazer o require do arquivo Endereco.php. Esse é um problema no qual trabalharemos no futuro. Por enquanto, resolveremos simplesmente fazendo require 'src/Endereco.php'.

require\_once 'src/Conta.php';

require\_once 'src/Titular.php';

require\_once 'src/CPF.php';

require\_once 'src/Endereco.php';

Entretanto, uma nova execução continuará apresentando um erro. Isso porque, como nosso Titular precisa de um Endereco, este último precisará ser importado primeiro. Trabalhoso, não? Teremos uma solução mais elegante para isso no futuro, mas por enquanto simplesmente mudaremos a ordem das importações.

require\_once 'src/Conta.php';

require\_once 'src/Endereco.php';

require\_once 'src/Titular.php';

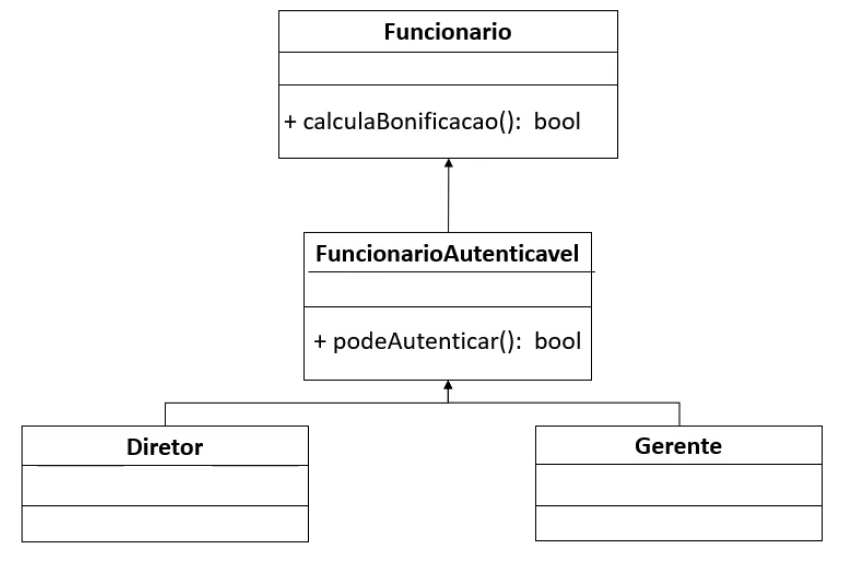
require\_once 'src/CPF.php';

Dessa vez a nossa execução acontecerá sem erros. Com isso, revisamos bastante conteúdo do curso anterior, como a criação de uma classe com atributos privados, a criação do construtor e dos métodos de acesso, a composição de objetos e assim por diante. No próximo vídeo começaremos a implementar novas funcionalidades.

**Heranças múltiplas**

Terminamos o vídeo anterior com uma nova demanda, na qual, além do Diretor, um Gerente também deve conseguir se autenticar. Sabemos que não é ideal dependermos do Funcionario para isso, pois implicaria na inclusão de diversos if no nosso código, mas a herança pode nos ajudar a resolver esse problema.

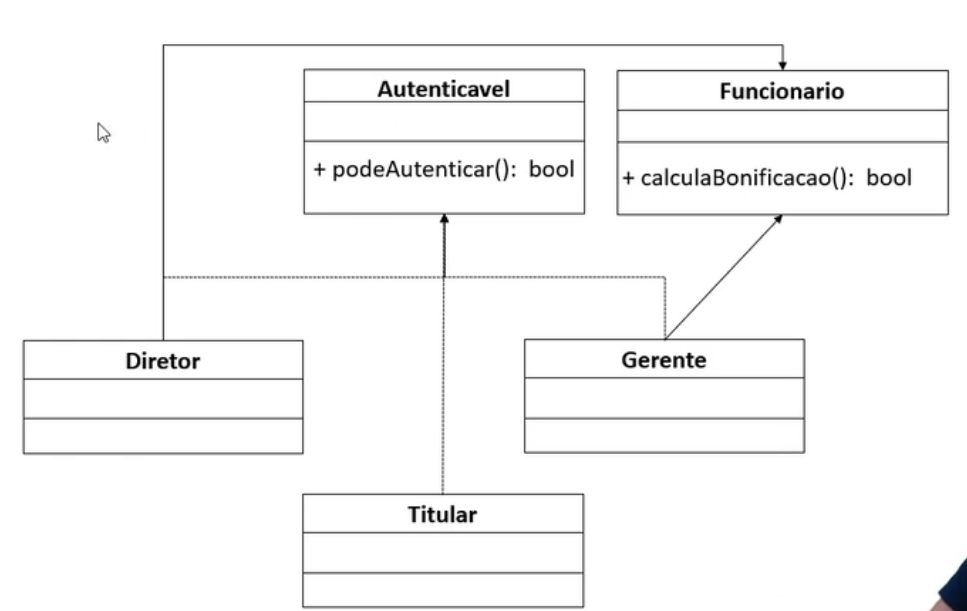
Por exemplo, podemos fazer com que Diretor e Gerente estendam de outra classe, chamada FuncionarioAutenticavel. Assim, evitaremos que o Desenvolvedor e o EditorVideo tenham acesso à autenticação.



Ou seja, além da classe Funcionario, teremos uma outra chamada FuncionarioAutenticavel possuindo um método podeAutenticar() que ficará acessível tanto a Diretor quanto a Gerente.

Agora imagine que temos uma nova demanda na qual o Titular de uma conta também deverá ter acesso ao sistema. Nesse caso, bastará fazermos com que essa classe também estenda de FuncionarioAutenticavel, certo? Na verdade isso não faz sentido, afinal o Titular não é um FuncionarioAutenticavel, nem mesmo necessariamente um Funcionario. Isso faria, por exemplo, com que o Titular tivesse um salário e passasse a receber bonificação. Ou seja, teremos que pensar em outra abordagem.

Uma ideia seria criarmos uma classe separada, chamada Autenticavel, que poderia ser estendida tanto por Titular quanto por Diretor e Gerente. A possibilidade de uma classe filha estender de mais de uma classe base/mãe é o que chamamos de **herança múltipla**. Nesse caso, Diretor e Gerente estenderão tanto de Autenticavel quanto de Funcionario.



Isso é possível em linguagens como o C++, mas não em outras, como Java e o próprio PHP. Mas se a herança múltipla parece algo tão útil, por que ela não existe em PHP?

Imagine que tenhamos na classe Funcionario um método chamado teste(), e outro com o mesmo nome na classe Autenticavel. Se Diretor herdar das duas classes, qual das duas implementações será herdada e executada quando o método teste() for chamado em uma instância de Diretor? Chamamos esse cenário de **problema diamante**, em referência ao desenho formado pela implementação lógica desse problema.

Como o PHP é uma das linguagens que não implementa a herança múltipla, teremos que encontrar outra forma de implementar um método podeAutenticar(), de maneira parecida com a existência de uma classe Autenticavel, mas mantendo somente a herança de Funcionario. Ou seja, a ideia é termos uma outra estrutura, que não é uma classe, firmando uma espécie de "*contrato*" que permitirá acesso ao método desejado. Entenderemos que estrutura é essa e como implementá-la no próximo vídeo.