



**INSTITUTO
FEDERAL**

Sudeste de
Minas Gerais

Campus
Manhuaçu

Desenvolvimento Web III

Introdução a programação orientada a objetos

Prof. Leonardo C. R. Soares - leonardo.soares@ifsudestemg.edu.br

Instituto Federal do Sudeste de Minas Gerais

21 de novembro de 2024



Introdução

Introdução

O conceito de orientação a objetos não é novo. Desde a década de 1960 já existiam linguagens que davam suporte a esta metodologia. Nos últimos anos, esse paradigma de programação tem ganhado cada vez mais destaque no mercado sendo utilizado por linguagens como **Java, JavaScript C++, PHP** (desde a versão 5) etc.

Orientação a Objetos

A OO - Orientação a Objetos - é também um termo geral que inclui qualquer estilo de desenvolvimento que seja baseado no conceito de *objeto* - uma entidade que exhibe características e comportamentos.





Classes e Objetos

Classe

Classe é uma representação abstrata de uma entidade existente no domínio do sistema. Funciona como um molde para a criação de objetos. Uma classe define todos os atributos (características) e métodos (comportamentos) que serão comuns a um grupo de objetos.





Atributos e métodos

Atributos

São as características que uma determinada classe possui. Os atributos são definidos pela classe, mas seus valores (em sua maioria) são definidos no objeto.

Métodos

São as ações que uma determinada classe pode executar.



Abstração

Definição

Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais.





Encapsulamento

Definição

Encapsulamento é uma característica que nos permite ocultar detalhes da implementação que não são necessários fora de um determinado contexto





Modificadores de acesso

Public, Protected e Private

- ▶ Public - O atributo ou método estará visível a todas as classes;
- ▶ Protected - O atributo ou método estará visível na classe onde foi declarado e nas subclasses dela;





Modificadores de acesso

Public, Protected e Private

- ▶ Public - O atributo ou método estará visível a todas as classes;
- ▶ Protected - O atributo ou método estará visível na classe onde foi declarado e nas subclasses dela;
- ▶ Private - O atributo ou método estará visível apenas na classe onde foi declarado.



Declaração de atributos

Sintaxe

Para declararmos atributos de classe, devemos utilizar o modificador de acesso seguido do nome do atributo. Caso o modificador de acesso seja omitido, será atribuído o modificador *public*.

```
class Animal{
    public string $nome;
    public int $idade;
}
```



Declaração de métodos

Sintaxe

Para declararmos métodos de classe, devemos utilizar o modificador de acesso seguido da palavra-chave *function*, do nome do método e, entre parênteses, os parâmetros que o método recebe. Caso o método não receba parâmetros, deve-se colocar parênteses vazios após o nome do método. Após os parâmetros, é possível definir o tipo de retorno do método, precedido por dois pontos.





Declaração de métodos

```
class Animal{  
    public string $nome;  
    public int $idade;  
    public function emitirSom(): void{  
        echo "...";  
    }  
    public function anoNascimento(int $anoAtual): int{  
        return $anoAtual - $this->idade;  
    }  
}
```

O operador `$this` é utilizado quando a classe precisa fazer uma referência ao objeto dela própria.

O operador `->` separa o objeto de seu atributo ou método.





Criação de objetos

Definição

Para criar um novo objeto, utilizamos a instrução *new* seguido do nome da classe.





Exercício Rápido

Desenvolva uma classe que possua dois atributos numéricos (num1 e num2) e métodos que permitam efetuar as quatro operações fundamentais sobre estes números. Instancie um objeto, dê valores aleatórios a num1 e num2 e imprima o resultado das quatro operações.





Exercício rápido - Correção

```
class Calculos {  
    public float $num1;  
    public float $num2;  
    public function somar(): float {  
        return $this->num1 + $this->num2; }  
    public function subtrair(): float{  
        return $this->num1 - $this->num2; }  
    public function dividir(): float{  
        return $this->num1 / $this->num2; }  
    public function multiplicar(): float{  
        return $this->num1 * $this->num2; }  
}
```





Exercício rápido - Correção

```
require_once "Calculos.php";  
$x = new Calculos();  
$x->num1 = rand(0,10);  
$x->num2 = rand(0,10);  
  
echo "Num1: $x->num1 - Num2: $x->num2<br>";  
echo $x->somar() . "<br>";  
echo $x->subtrair() . "<br>";  
echo $x->multiplicar() . "<br>";  
echo $x->dividir() . "<br>";
```





Exercício rápido - Correção

```
require_once "Calculos.php";  
$x = new Calculos();  
$x->num1 = rand(0,10);  
$x->num2 = rand(0,10);  
  
echo "Num1: $x->num1 - Num2: $x->num2<br>";  
echo $x->somar() . "<br>";  
echo $x->subtrair() . "<br>";  
echo $x->multiplicar() . "<br>";  
echo $x->dividir() . "<br>";
```

O que acontece se o num2 for zero?





Exercício rápido - Correção

```
require_once "Calculos.php";  
$x = new Calculos();  
$x->num1 = rand(0,10);  
$x->num2 = rand(0,10);  
  
echo "Num1: $x->num1 - Num2: $x->num2<br>";  
echo $x->somar() . "<br>";  
echo $x->subtrair() . "<br>";  
echo $x->multiplicar() . "<br>";  
echo $x->dividir() . "<br>";
```

O que acontece se o num2 for zero? Para corrigir devemos lançar uma exceção e tratá-la.





Exercício rápido - Correção

```
class Calculos {  
    public float $num1;  
    public float $num2;  
    public function somar(): float {  
        return $this->num1 + $this->num2; }  
    public function subtrair(): float{  
        return $this->num1 - $this->num2; }  
    public function dividir(): float{  
        if ($this->num2 == 0)  
            throw new Exception("Impossível dividir por 0.");  
        return $this->num1 / $this->num2; }  
    public function multiplicar(): float{  
        return $this->num1 * $this->num2; }  
}
```





Exercício rápido - Correção

```
$x = new Calculos();  
$x->num1 = rand(0,10);  
$x->num2 = rand(0,10);  
echo "Num1: $x->num1 - Num2: $x->num2<br>";  
echo $x->somar() . "<br>";  
echo $x->subtrair() . "<br>";  
echo $x->multiplicar() . "<br>";  
try{  
    echo $x->dividir() . "<br>";  
}catch(Exception $e){  
    echo $e->getMessage();  
}
```





Encapsulamento: Como fazer

O básico do encapsulamento é feito restringindo-se o acesso direto aos atributos da classe. Isto deve ser feito para garantir que as regras de negócio da classe, desconhecidas por usuários do objeto, não sejam violadas.





Encapsulamento: Como fazer

O básico do encapsulamento é feito restringindo-se o acesso direto aos atributos da classe. Isto deve ser feito para garantir que as regras de negócio da classe, desconhecidas por usuários do objeto, não sejam violadas.

Para permitir que usuários do objeto possam, quando necessário, definir ou ler os valores dos atributos, devem ser criados métodos especiais denominados **método acessores**.





Encapsulamento: Métodos acessores

Métodos acessores responsáveis por retornar o valor de atributos devem ser nomeados como **getNomeAtributo**. Em geral estes métodos não recebem parâmetros e não fazem validações.





Encapsulamento: Métodos acessores

Métodos acessores responsáveis por retornar o valor de atributos devem ser nomeados como **getNomeAtributo**. Em geral estes métodos não recebem parâmetros e não fazem validações.

Métodos acessores responsáveis por atribuir valores a atributos devem ser nomeados como **setNomeAtributo**. Em geral estes métodos recebem como parâmetro o valor a ser atribuído e, caso existam regras de negócio sobre os valores possíveis, elas são aplicadas antes da atribuição dos mesmos.





Encapsulamento: Exemplo

Considere uma classe para representar um funcionário. Esta classe tem os atributos nome e salário. Salário não pode ser negativo.





Encapsulamento: Exemplo

```
<?php
class Funcionario{
    private string $nome;
    private float $salario;
    public function getNome():string{
        return $this->nome;
    }
    public function setNome($nome):void{
        $this->nome = $nome;
    }
    public function getSalario():float{
        return $this->salario;
    }
    public function setSalario($salario):void{
        if ($salario<0)
            throw new Exception("Salário não pode ser negativo.");
        $this->salario=$salario;
    }
}
?>
```





Membros estáticos

Definição

Membros de classe estáticos são atributos ou métodos que pertencem a classe e não ao objeto. Não importa a quantidade de instâncias que uma classe possua, há apenas uma cópia de cada membro estático.





Membros estáticos

Declaração e uso

Para declarar um membro como estático, utilizamos a palavra-chave *static*.





Membros estáticos

Declaração e uso

Para declarar um membro como estático, utilizamos a palavra-chave *static*.

Quando a própria classe deseja acessar um membro estático, deve-se utilizar a palavra-chave *self* seguido de `::`





Membros estáticos

Declaração e uso

Para declarar um membro como estático, utilizamos a palavra-chave *static*.

Quando a própria classe deseja acessar um membro estático, deve-se utilizar a palavra-chave *self* seguido de `::`

Quando alguém fora da classe deseja acessar um membro estático, deve utilizar o nome da classe seguido de `::`





Membros estáticos

Declaração e uso

Para declarar um membro como estático, utilizamos a palavra-chave *static*.

Quando a própria classe deseja acessar um membro estático, deve-se utilizar a palavra-chave *self* seguido de `::`

Quando alguém fora da classe deseja acessar um membro estático, deve utilizar o nome da classe seguido de `::`

Um método estático não pode acessar uma propriedade não estática (dinâmica).





Exemplo

```
class Circulo {  
    public static float $PI = 3.1415;  
    public static function calculaArea(float $raio): float {  
        return self::$PI*($raio*$raio);  
    }  
}  
  
echo "O valor de PI é ". Circulo::$PI;  
echo "A área de um círculo com 10cm de raio é ".Circulo::calculaArea(10);
```





Métodos construtores

Definição

Métodos construtores definem quais ações devem ser executadas quando da criação de um objeto. Implicitamente, um método construtor retorna um objeto do próprio tipo da classe.





Métodos construtores

Definição

Métodos construtores definem quais ações devem ser executadas quando da criação de um objeto. Implicitamente, um método construtor retorna um objeto do próprio tipo da classe.

Em PHP, o método construtor de uma classe é nomeado como **__construct**. Assim como qualquer outro método, o construtor pode receber parâmetros. Caso isso aconteça, os parâmetros devem ser informados na criação do objeto.





Exemplo

```
class Funcionario{
    private string $nome;
    private float $salario;
    public function __construct(string $nome, float $sal){
        $this->setNome($nome);
        $this->setSalario($sal);
    }
    public function getName():string{
        return $this->nome;
    }
    public function setNome($nome):void{
        $this->nome = $nome;
    }
    public function getSalario():float{
        return $this->salario;
    }
    public function setSalario($salario):void{
        if ($salario<0)
            throw new Exception("Salário não pode ser negativo.");
        $this->salario=$salario;
    }
}

$p = new Funcionario("Maria",1780);
```





Métodos destrutores

Definição

Métodos destrutores são invocados quando todas as referências a um objeto particular forem removidas ou quando o objeto for explicitamente destruído.





Métodos destrutores

Definição

Métodos destrutores são invocados quando todas as referências a um objeto particular forem removidas ou quando o objeto for explicitamente destruído.

Em PHP, o método destrutor de uma classe é nomeado como **`__destruct`**. Assim como qualquer outro método, o destrutor pode receber parâmetros. Caso isso aconteça, os parâmetros devem ser informados na criação do objeto.



Exemplo

```
<?php
class LOG {
    public $arquivo;
    public function __construct(){
        $this->arquivo = fopen("log.txt","a");
        fwrite($this->arquivo,"Usuário logou em " . date("d/m/Y H:i:s") . "\n");
    }

    public function __destruct(){
        fwrite($this->arquivo,"Usuário saiu em " . date("d/m/Y H:i:s") . "\n");
        fclose($this->arquivo);
    }
}

$log = new Log();

?>
```





Sobrecarga

Atenção:

PHP define como sobrecarga a criação dinâmica de atributos e métodos. **Não é o nosso enfoque.** Estamos tratando sobrecarga sobre a ótica da programação orientada à objetos.



Herança

Definição

Herança é um recurso da POO que nos permite criar novas classes a partir de classes já existentes.



Herança

Definição

Herança é um recurso da POO que nos permite criar novas classes a partir de classes já existentes.

A classe de quem herdaremos denomina-se **classe base** ou **super classe**. Em geral, ela representa um conceito mais genérico.





Herança - Exemplo

```
<?php
class PessoaFisica {
    protected string $nome;
    protected string $cpf;
    public function getNome(): string { return $this->nome; }
    public function setNome(string $n){ $this->nome = $n; }
    public function getCpf(): string { return $this->cpf; }
    public function setCpf(string $cpf){ $this->cpf = $cpf; }
}
?>
```





Herança - Exemplo

```
require_once "PessoaFisica.class.php";  
class Funcionario extends PessoaFisica {  
    private string $cargo;  
    private float $salario;  
    public function setCargo($c){ $this->cargo = $c;}  
    public function getCargo(): string { return $this->cargo;}  
    public function setSalario($s){ $this->salario = $s;}  
    public function getSalario():float{ return $this->salario*0.89;}  
}
```



Herança - Exemplo

```
$f = new Funcionario();
$f->setNome("Mariana");
$f->setCpf("123456");
$f->setCargo("Professora");
$f->setSalario(5000);

echo "Olá " . $f->getNome() . " seu salário líquido é R$ " . $f->getSalario();
?>
```





Herança - Exemplo

```
class PessoaFisica {
    protected string $nome;
    protected string $cpf;
    public function __construct(string $cpf, string $nome){
        $this->setNome($nome);
        $this->setCpf($cpf);
    }

    public function getNome(): string { return $this->nome; }
    public function setNome(string $n){ $this->nome = $n; }
    public function getCpf(): string { return $this->cpf; }
    public function setCpf(string $cpf){ $this->cpf = $cpf; }
}
```





Herança - Exemplo

```
require_once "PessoaFisica.class.php";
class Funcionario extends PessoaFisica {
    private string $cargo;
    private float $salario;
    public function __construct(string $cpf, string $nome, string $cargo, float $salario){
        parent::__construct($cpf, $nome);
        $this->setCargo($cargo);
        $this->setSalario($salario);
    }
    public function setCargo($c){ $this->cargo = $c;}
    public function getCargo(): string { return $this->cargo;}
    public function setSalario($s){ $this->salario = $s;}
    public function getSalario():float{ return $this->salario*0.89;}
}
```



Herança - Exemplo

```
$f = new Funcionario("1234", "Mariana", "Professora", 5000);  
echo "Olá ". $f->getCargo() . " ". $f->getNome() . " seu salário líquido é R$ ".$f->getSalario();
```



Exemplo

```
require_once "Funcionario.class.php";  
class Gerente extends Funcionario {  
    public function __construct(string $cpf, string $nome, float $salario){  
        parent::__construct($cpf,$nome,"Gerente",$salario);  
    }  
  
    public function getSalario(): float{  
        $salarioLiquido = parent::getSalario();  
        return $salarioLiquido*1.15;  
    }  
}
```



Exemplo

```
$f = new Funcionario("1234","Mariana", "Professora", 5000);
$g = new Gerente("2333","Júlia",5000);
echo "Olá ". $f->getCargo() ." ". $f->getNome() . " seu salário líquido é R$ ".$f->getSalario();
echo "<br>";
echo "Olá ". $g->getCargo() ." ". $g->getNome() . " seu salário líquido é R$ ".$g->getSalario();
?>
```



Exemplo

```
abstract class Animal{
    public string $nomeCientifico;
    public abstract function emitirSom(): string;
    public abstract function dormir(): string;
    public function setNomeCientifico($n):void{
        $this->nomeCientifico = $n;
    }
    public function getNomeCientifico():string {
        return $this->nomeCientifico;
    }
}
```





Exemplo

```
abstract class Animal{  
    public string $nomeCientifico;  
    public abstract function emitirSom(): string;  
    public abstract function dormir(): string;  
    public function setNomeCientifico($n):void{  
        $this->nomeCientifico = $n;  
    }  
    public function getNomeCientifico():string {  
        return $this->nomeCientifico;  
    }  
}
```

Se tentarmos criar um objeto desta classe (new Animal), receberemos o erro: Fatal error: Uncaught Error: Cannot instantiate abstract class Animal in ...



Subclasses concretas

Classes concretas que herdam de classes abstratas devem, obrigatoriamente, implementar os métodos abstratos que foram herdados. No exemplo apresentado, toda classe concreta que herdar de `Animal` (é-um) deve implementar os métodos `emitirSom()` e `dormir()`.



Exemplo

```
class Gato extends Animal {
    public function emitirSom(): string {
        return "miau";
    }
    public function dormir(): string {
        return "zzzZZzZZZ";
    }
}
```



Exemplo

```
class Base {
    public final function sayHello(): void{
        echo "hello";
    }
}

class Filha extends Base {
    public function sayHello(): void{
        echo "Olá";
    }
}
```

Fatal error: Cannot override final method Base::sayHello() in...



Exemplo

```
final class Base {  
    // 0 final aqui é indiferente  
    public final function sayHello(): void{  
        echo "hello";  
    }  
}  
  
class Filha extends Base {  
    public function sayHello(): void{  
        echo "Olá";  
    }  
}
```

Fatal error: Class Filha may not inherit from final class (Base) in...



Interfaces

Definição

Podemos entender a *interface de uma classe* como sendo o conjunto de comportamentos que esta classe é capaz de executar.



Interfaces

Definição

Podemos entender a *interface de uma classe* como sendo o conjunto de comportamentos que esta classe é capaz de executar.

Assim, uma Interface especifica o comportamento esperado para um conjunto de classes que a implementam.



Interfaces

Definição

Podemos entender a *interface de uma classe* como sendo o conjunto de comportamentos que esta classe é capaz de executar.

Assim, uma Interface especifica o comportamento esperado para um conjunto de classes que a implementam.

Diversos autores definem interface como um contrato a ser seguido por um conjunto de classes (as classes que implementam a interface).





Interfaces

Definição

Podemos entender a *interface de uma classe* como sendo o conjunto de comportamentos que esta classe é capaz de executar.

Assim, uma Interface especifica o comportamento esperado para um conjunto de classes que a implementam.

Diversos autores definem interface como um contrato a ser seguido por um conjunto de classes (as classes que implementam a interface).

Interfaces definem o que um conjunto de classes (que implementam a interface) devem fazer e **não como** deve ser feito.



Interfaces

Definição

A criação de interfaces em PHP seguem o mesmo padrão estabelecido para a criação de classes, apenas trocamos a palavra-chave `class` por `interface`.

Todos os métodos declarados em uma interface são públicos e implicitamente abstratos. A palavra-chave `abstract` não deve ser colocada nos métodos (para interfaces).



Exemplo

```
1 interface A{
2     public function foo(): void;
3 }
```



Relacionamento entre classes e interfaces

Classes implementam interfaces

Ao implementar uma interface, uma classe concreta é obrigada a implementar **todos** os métodos declarados na interface. O relacionamento é similar ao que ocorre quando uma classe concreta herda de uma classe abstrata. Entretanto, como não se trata de um relacionamento de herança, é possível que uma mesma classe implemente **diversas** interfaces.



Polimorfismo

```
interface Animal{
    public function emitirSom(): string;
}

class Cachorro implements Animal {
    public function emitirSom(): string {
        return "au au au"; }
}

class Gato implements Animal {
    public function emitirSom(): string {
        return "miau"; }
}

class Lobo implements Animal {
    public function emitirSom(): string {
        return "auuuu"; }
}

class Ornitorrinco implements Animal {
    public function emitirSom(): string {
        return "arghhh"; }
}
```



