SW-I
SISTEMAS WEB I

Prof. Anderson Vanin

AULA 11 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM PHP

MÉTODO CONSTRUTOR, ENCAPSULAMENTO E GETTERS E SETTERS

Objetivos

- Compreender o papel do método construtor (**`__construct`**).
- Aprender a aplicar encapsulamento usando **`private`**, **`public`** e **`protected`**.
- Utilizar **`getters`** e **`setters`** para acessar atributos privados.
- Desenvolver pequenos sistemas orientados a objetos.

Introdução ao Método Construtor

Em PHP, o método construtor é uma função especial com o nome **__construct**. Ele é automaticamente chamado quando criamos um novo objeto a partir de uma classe. Sua principal função é inicializar o objeto com valores iniciais, logo no momento da criação.

```php
$usuario = new Usuario("João", "joao@email.com");
```

Por que usar construtores é uma boa prática?

1. **Organização do Código**

O construtor centraliza a lógica de inicialização dos objetos. Isso significa que qualquer instância criada terá um ponto único de entrada para configurar seus valores iniciais.

🔍 Sem construtor:

```php
$produto = new Produto();
$produto->setNome("Notebook");
$produto->setPreco(3500);
```

🔍 Com construtor:

```php
$produto = new Produto("Notebook", 3500);
```

Por que usar construtores é uma boa prática?

2. **Obrigatoriedade de Dados**

Ao exigir parâmetros no construtor, forçamos o programador a fornecer dados importantes no momento da criação do objeto. Isso evita objetos malformados ou com atributos vazios.

```php
class Usuario {
    private $nome;

    public function __construct($nome) {
        $this->nome = $nome;
    }
}
```

Se alguém tentar instanciar **`new Usuario();`**, haverá um erro, alertando que um nome é necessário.

Por que usar construtores é uma boa prática?

3. **Segurança e Encapsulamento**

Construtores ajudam a respeitar o princípio de encapsulamento. Mesmo com atributos **private**, conseguimos configurar os dados de forma controlada logo na criação. Isso impede que atributos internos sejam manipulados diretamente, e ainda garante que todos os objetos iniciem em um estado válido.

Por que usar construtores é uma boa prática?

4. **Flexibilidade com Valores Opcionais**

É possível usar valores padrão em parâmetros do construtor, o que torna a criação de objetos ainda mais flexível:

```php
class Produto {
    public function __construct($nome, $preco = 0.0, $quantidade = 0) {
        // ...
    }
}

$p1 = new Produto("Caneta");
$p2 = new Produto("Caderno", 15.90, 10);
```

Por que usar construtores é uma boa prática?

5. **Facilidade para Manutenção e Expansão**

Quando novas propriedades precisam ser adicionadas a uma classe, o construtor pode ser facilmente adaptado. Isso ajuda na manutenção e evolução do código. Além disso, facilita testes e instância de objetos em outros contextos, como APIs, sistemas web ou testes automatizados.

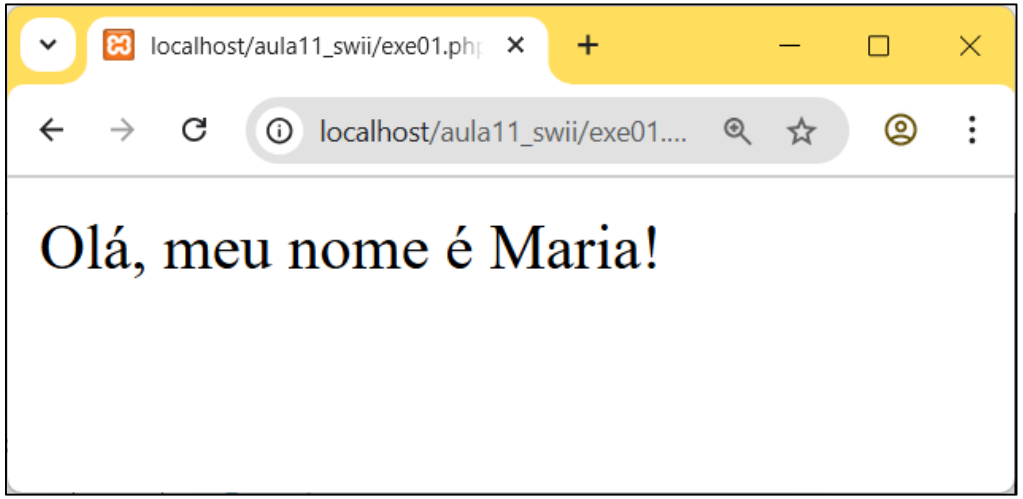
Exemplo

Pessoa.class.php

```php
<?php
    class Pessoa {
        private $Nome;

        public function __construct($nome_recebido) {
            $this->Nome = $nome_recebido;
        }

        public function Apresentar() {
            echo "Olá, meu nome é {$this->Nome}!";
        }
}

?>
```

exe01.php

```php
<?php
    include_once 'Pessoa.class.php';
    $p1 = new Pessoa("Maria");
    $p1->Apresentar();

?>
```



Encapsulamento com Getters e Setters

O que é Encapsulamento?

Encapsulamento é um princípio fundamental da programação orientada a objetos. Ele define que os atributos de uma classe devem ser protegidos do acesso direto externo, permitindo o controle desse acesso através de métodos públicos: os famosos ***getters*** e ***setters***.

Por que encapsular os atributos é uma boa prática?

1. **Proteção dos Dados Internos**

Ao tornar os atributos **private**, evitamos que outras partes do sistema possam alterá-los de forma direta e descontrolada.

🔴 Exemplo ruim (sem encapsulamento):

```php
$carro->velocidade = -200; // Isso não deveria ser permitido!
```

🟢 Com encapsulamento:

```php
$carro->setVelocidade(-200); // Podemos validar dentro do método!
```

Encapsulando, conseguimos **controlar, validar e proteger** os dados internos do objeto.

Por que encapsular os atributos é uma boa prática?

2. **Maior Controle com Validações**

Getters e **Setters** não servem apenas para ler e modificar valores, mas também para aplicar lógicas de controle.

```php
public function setIdade($idade) {
    if ($idade >= 0) {
        $this->idade = $idade;
    } else {
        echo "Idade inválida!";
    }
}
```

Essa abordagem **evita que o objeto fique em um estado inválido**, mantendo a integridade dos dados.

Por que encapsular os atributos é uma boa prática?

3. **Abstração e Flexibilidade**

O encapsulamento permite mudar a forma como os dados são manipulados, sem alterar quem usa a classe. Por exemplo:

```php
public function getNomeCompleto() {
    return $this->nome . " " . $this->sobrenome;
}
```

Aqui, o código externo não sabe se os nomes estão salvos separadamente. Isso cria uma **abstração**: quem usa o objeto só se preocupa com a interface, não com a implementação.

Por que encapsular os atributos é uma boa prática?

4. **Manutenção e Evolução Seguras**

Com atributos privados, é possível alterar a estrutura interna da classe sem afetar os sistemas que a utilizam, desde que os métodos públicos permaneçam consistentes. Isso torna a manutenção mais simples e segura em sistemas grandes e duradouros.

5. **Aderência aos Princípios de Engenharia de Software**

Encapsulamento favorece o princípio da responsabilidade única e o princípio do mínimo conhecimento: cada classe conhece apenas o necessário e não permite interferência direta de outras partes do sistema.

Encapsulamento com Getters e Setters

Reforçamos que atributos **private** não podem ser acessados diretamente fora da classe. Para isso, usamos métodos:

- **`getAtributo()`** → para ler o valor.
- **`setAtributo()`** → para modificar o valor.

Exemplo Integrado: Classe **ContaBancaria**

exemplo_contabancaria > Conta.class.php

```php
<?php
    class ContaBancaria {
        private $Titular;
        private $Saldo;

        // Construtor para iniciar titular e saldo inicial
        public function __construct($titular, $saldoInicial = 0) {
            $this->Titular = $titular;
            $this->Saldo = $saldoInicial;
        }

        // Getters
        public function getTitular() {
            return $this->Titular;
        }

        public function getSaldo() {
            return $this->Saldo;
        }

        // Métodos de operação
        public function Depositar($valor) {
            if ($valor > 0) {
                $this->Saldo += $valor;
            }
        }

        public function Sacar($valor) {
            if ($valor > 0 && $valor <= $this->Saldo) {
                $this->Saldo -= $valor;
            } else {
                echo "Saque não autorizado.\n";
            }
        }
    }
?>
```

Exemplo Integrado: Classe **ContaBancaria**

exemplo_contabancaria > Conta.class.php

```php
<?php
    class ContaBancaria {
        private $Titular;
        private $Saldo;

        // Construtor para iniciar titular e saldo inicial
        public function __construct($titular, $saldoInicial = 0) {
            $this->Titular = $titular;
            $this->Saldo = $saldoInicial;
        }

        // Getters
        public function getTitular() {
            return $this->Titular;
        }

        public function getSaldo() {
            return $this->Saldo;
        }

        // Métodos de operação
        public function Depositar($valor) {
            if ($valor > 0) {
                $this->Saldo += $valor;
            }
        }

        public function Sacar($valor) {
            if ($valor > 0 && $valor <= $this->Saldo) {
                $this->Saldo -= $valor;
            } else {
                echo "Saque não autorizado.\n";
            }
        }
    }
?>
```

Exemplo Integrado: Classe **ContaBancaria**

exemplo_contabancaria > conta.php

```php
<?php
    include_once 'Conta.class.php';
    $conta = new ContaBancaria("João", 1000);
    $conta->depositar(500);
    $conta->sacar(200);
    echo "Titular: " . $conta->getTitular() . "\n";
    echo "Saldo: R$" . $conta->getSaldo() . "\n";

?>
```

Exercício Prático

Criar uma classe **`Produto`** com uso de **construtor e encapsulamento**.

Instruções:

1. Crie a classe Produto **com os atributos privados**:

- **`Nome`**
- **`Preco`**
- **`Quantidade`**

2. Use um construtor para inicializar esses atributos. Implemente os métodos:

- **`getNome(), setNome()`**
- **`getPreco(), setPreco()`**
- **`adicionarEstoque($qtd)`**
- **`removerEstoque($qtd)`**
- **`mostrarDetalhes()`**

Exercício Prático

Exemplo de uso esperado:

```php
$produto = new Produto("Mouse", 150.00, 10);
$produto->adicionarEstoque(5);
$produto->removerEstoque(3);
$produto->mostrarDetalhes();
```

Saída esperada:

```
Produto: Mouse
Preço: R$150.00
Quantidade em estoque: 12
```