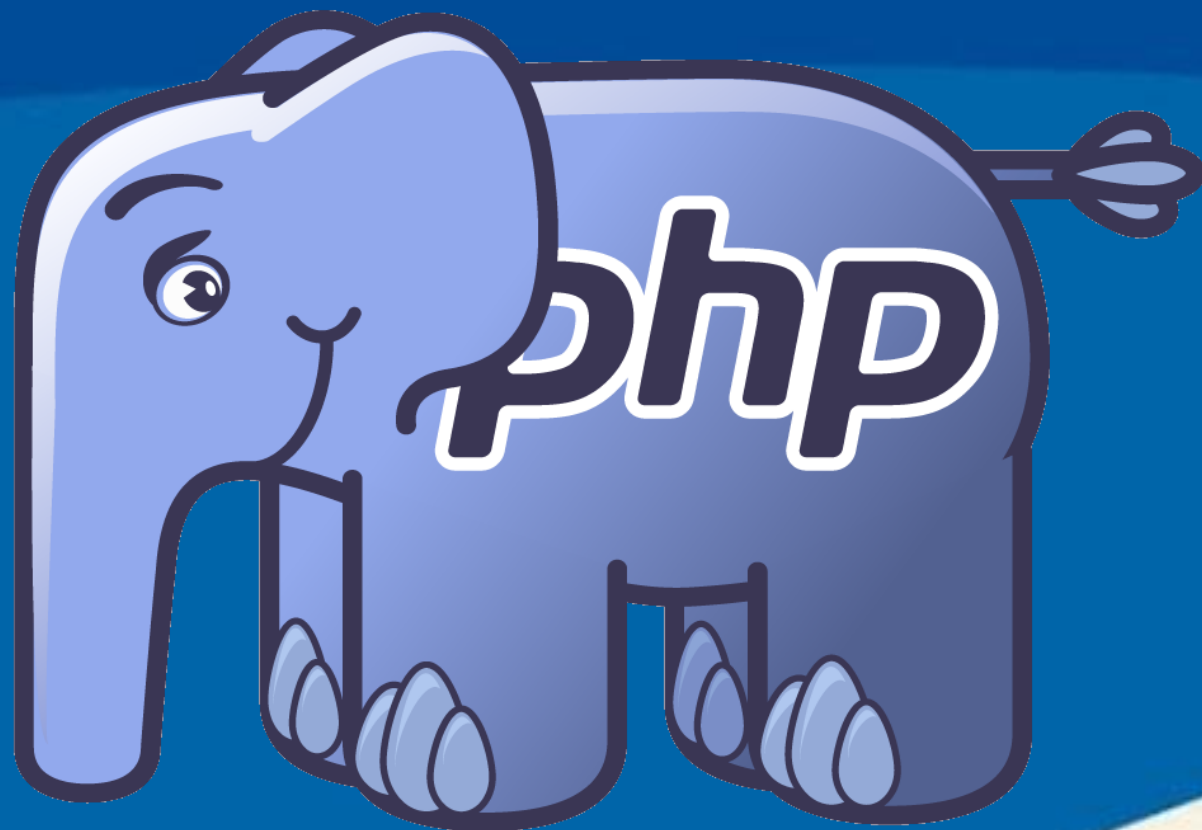


Programação Orientada a Objetos

Professor: Eduardo Florence



Introdução



A engenharia de software surgiu no final da década de 60 para tentar solucionar os problemas gerado pela “Crise de software”, no entanto, várias técnicas que foram desenvolvidas nos anos 70 e 80 não conseguiam resolver os problemas de produtividade e qualidade de software.

Introdução



Na década de 90 o computador passou a ser uma ferramenta de trabalho para muitas pessoas .

Com tudo isso, surge a necessidade de se produzir softwares mais atraentes dinâmicos e com alto poder de troca de informação, além disso, passou-se a exigir dos softwares uma maior produtividade e melhor qualidade.

Introdução

Através de pesquisas foi verificado que a reutilização de peças chaves para melhorar a produtividade e a qualidade de software era a solução para os problemas.



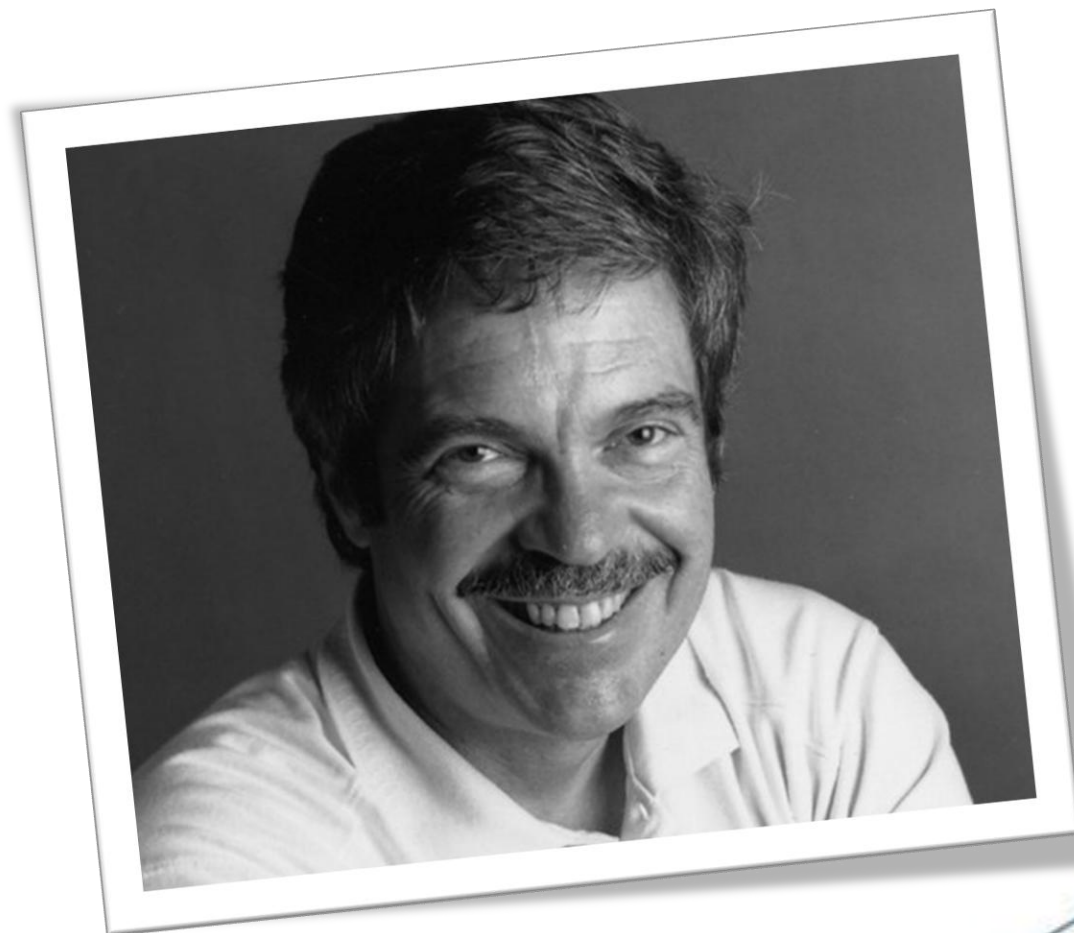
PROGRAMAÇÃO ORIENTADA A OBJETOS



As técnicas oferecidas pela programação estruturada não eram suficientes para atender, com satisfação desejada, a elaboração desses tipos de aplicações. Seria necessário partir para uma nova técnica de programação. Assim surgiu a técnica de orientação a objetos.

COMO SURTIU?

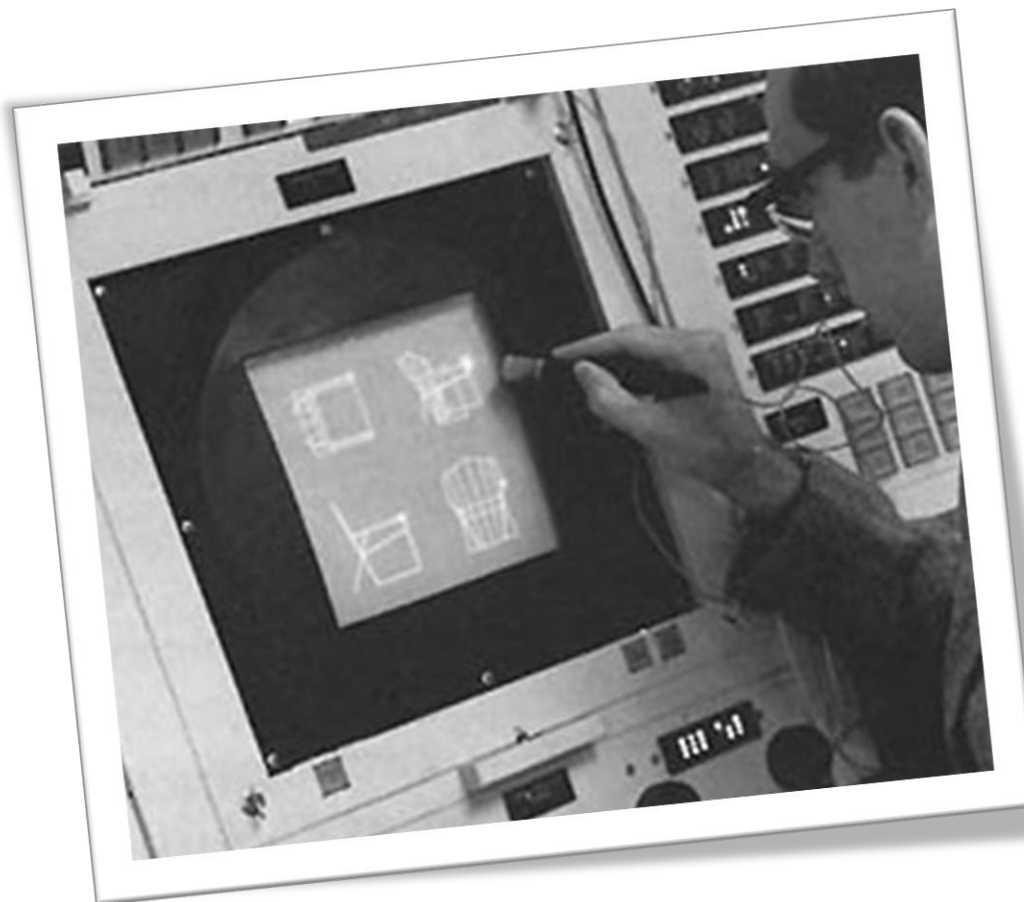
A orientação a objetos foi desenvolvida no laboratório da Xerox, em Palo Alto, em uma linguagem chamada Smalltalk. O líder desse projeto foi Alan Curtis Kay, considerado um dos criadores do termo “programação orientada a objetos”.



COMO SURTIU?

Alan Kay começou a programar em **Simula**, em um editor gráfico, chamado Sketchpad.

A partir dos conceitos desse sistema, e através de seus conhecimentos em Biologia e Matemática, Alan Kay formulou sua analogia “algébrico-biológica”.



ANALOGIA PERFEITA

Para Alan, o computador ideal deveria funcionar como um organismo vivo, isto é, cada “célula” deveria comportar-se relacionando com outras células a fim de alcançar um objetivo e de forma autônoma.

As células poderiam também reagrupar-se para resolver outros problemas ou desempenhar outras funções, trocando mensagens “químicas” entre elas.



PRINCÍPIOS DA ORIENTAÇÃO A OBJETOS



Alan Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagissem entre si, estabelecendo os seguintes princípios da orientação a objetos:

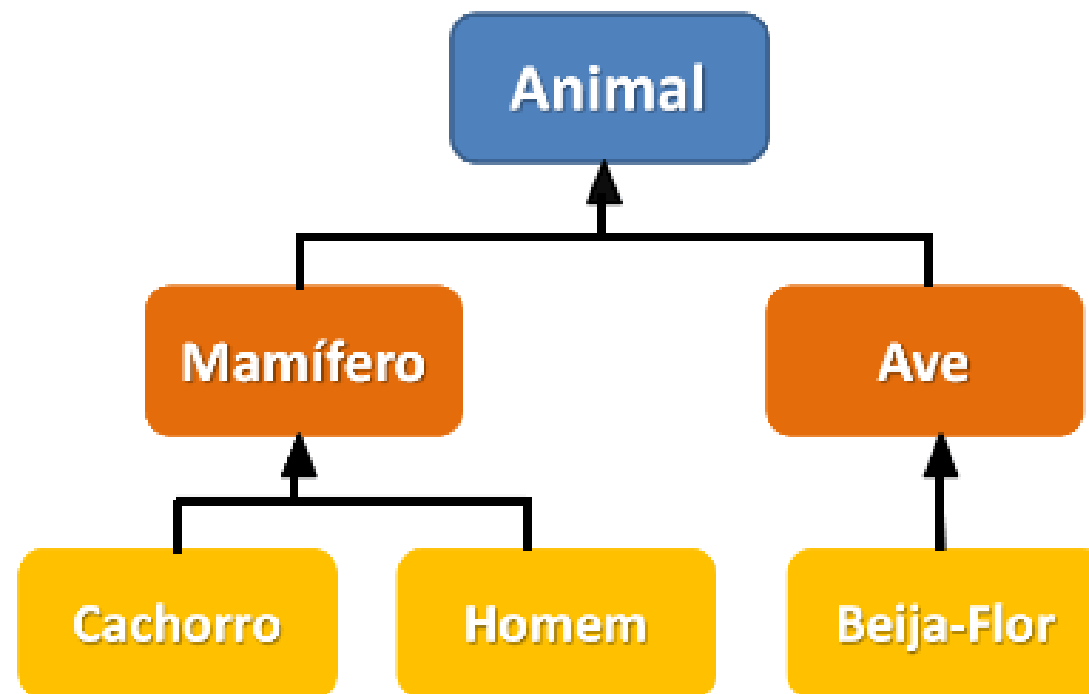
- Qualquer coisa é um objeto;
- Objetos realizam tarefas através da requisição de serviços;
- Cada objeto pertence a uma determinada classe;
- Uma classe agrupa objetos similares;
- Um classe possui comportamentos associados ao objeto; e
- Classes são organizadas em hierarquias.

PARADIGMA

Portando, depois de reunir conceitos de diversas áreas do conhecimento e com base em sua experiência como pesquisador, Alan Kay criou o paradigma de análise e programação mais eficiente da atualidade. Conhecer a origem da orientação a objetos ajuda a compreender como utilizá-la no desenvolvimento de projetos de software.

O QUE É POO?

A programação Orientada a objetos (POO) é uma forma especial de programar, mais próximo de como expressaríamos as coisas na vida real do que outros tipos de programação.



BENEFÍCIOS EM PROGRAMAR EM POO



- ✓ **Reutilização de código:** Evita a duplicação de código.
- ✓ **Manutenção:** Facilita a identificação e correção de erros.
- ✓ **Modularidade:** Permite dividir o sistema em partes menores e mais gerenciáveis.
- ✓ **Extensibilidade:** Facilita a adição de novas funcionalidades.

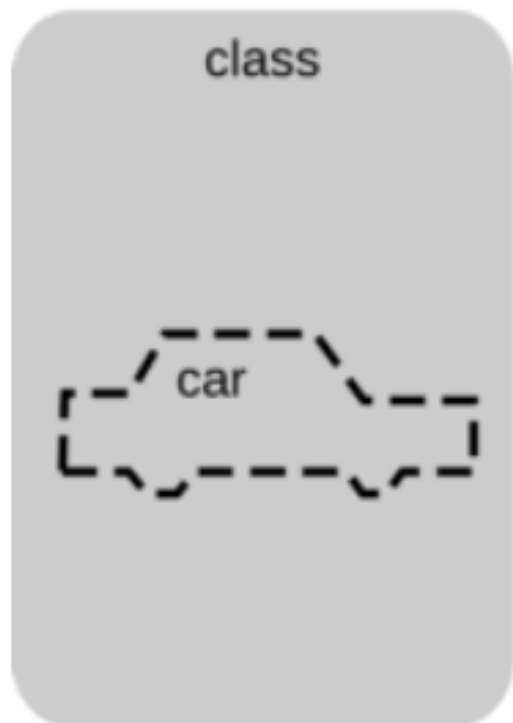
Ao compreender e aplicar esses pilares, você estará mais preparado para desenvolver aplicações PHP robustas, escaláveis e de alta qualidade.

ELEMENTOS DA POO

A Programação Orientada a Objetos é formada por alguns itens, dentre os quais destacamos:

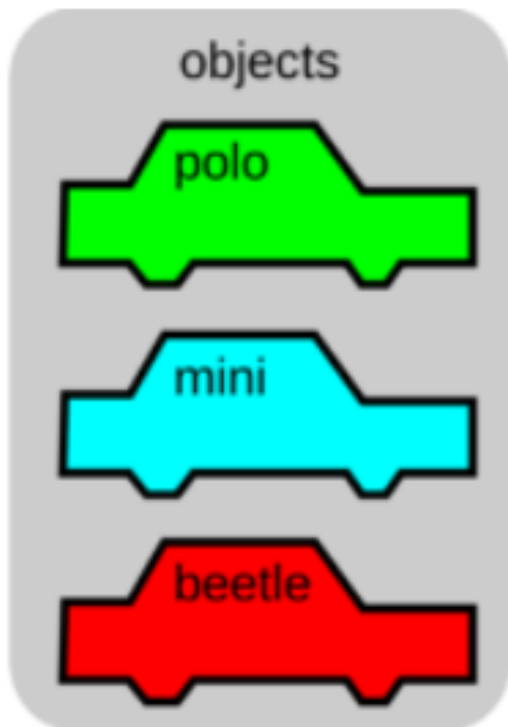


CLASSE



- É uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos - através de **métodos** - e os estados possíveis destes objetos - através de **atributos**.
- As classes são as que dão origem a objetos. Ou, ainda, são as que definem novos (e personalizados) “tipos de dados”.

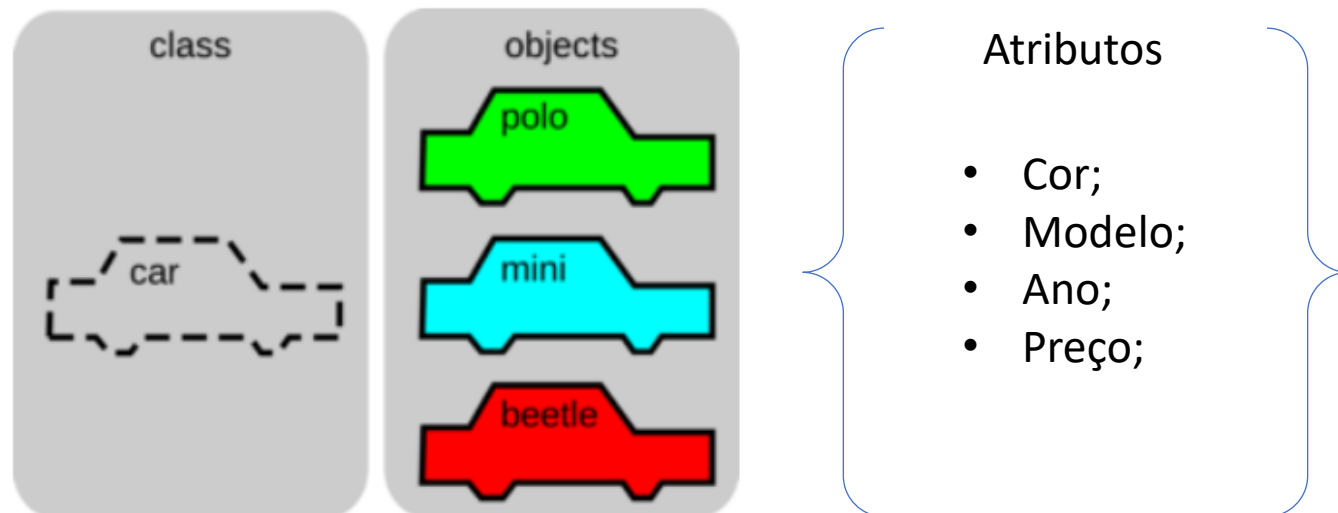
OBJETO



- Um **objeto**, na vida real, é qualquer coisa a qual pudermos tocar, manipular e dar um nome.
- Um **objeto**, em programação orientada a objetos, é uma instância, ou seja, um exemplar de uma classe.

ATRIBUTOS

Os atributos são as propriedades de um objeto, também são conhecidos como variáveis ou campos. Essas propriedades definem o estado de um objeto, fazendo com que esses valores possam sofrer alterações.



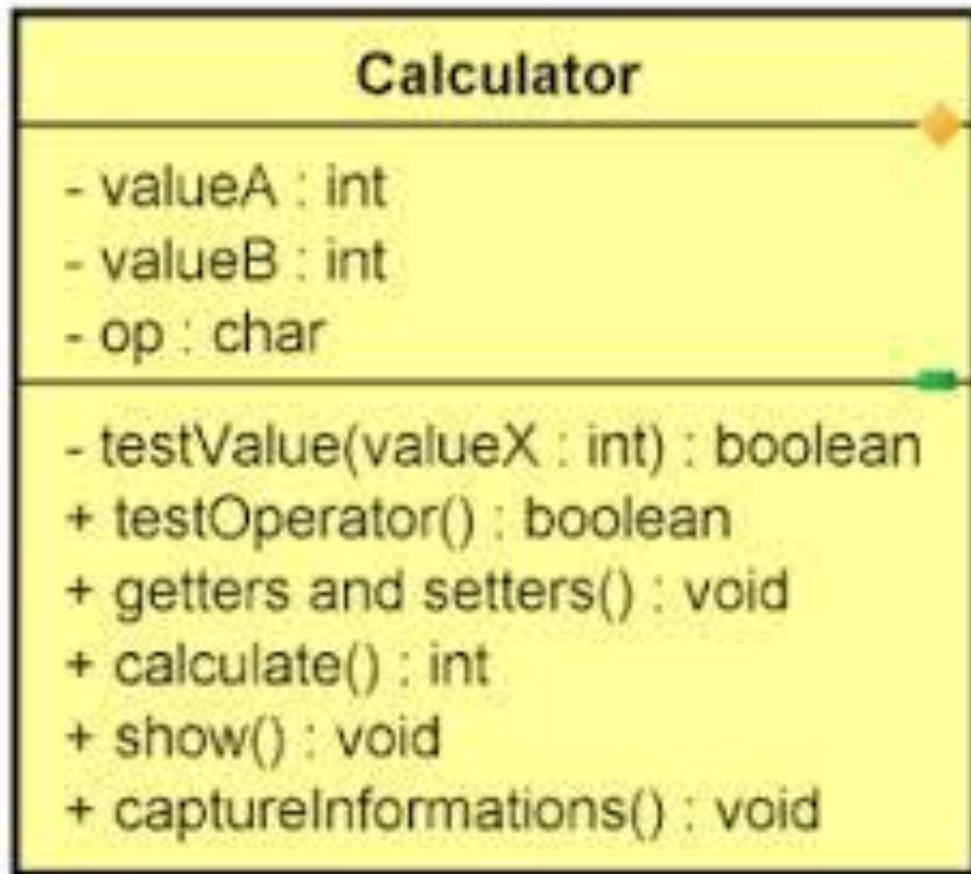
MÉTODO

Os métodos são ações ou procedimentos, onde podem interagir e se comunicarem com outros objetos. A execução dessas ações se dá através de mensagens, tendo como função o envio de uma solicitação ao objeto para que seja efetuada a rotina desejada.

Métodos

- Acelerar();
- Frear();
- Ligar();
- Desligar();

UML



Classe

- Atributos.
- Métodos.

EXERCÍCIOS POO – PARTE I

Em uma folha a parte apresente os **atributos** que as classes abaixo poderiam ter:



Carro



Cachorro



Pessoa

EXERCÍCIOS POO – PARTE II

Em uma folha a parte apresente um **método** que as classes abaixo poderiam ter:



Carro



Cachorro



Pessoa

EXERCÍCIOS POO – PARTE III

Crie uma classe chamada "Carro".

Essa classe deve ter os seguintes atributos:

- marca (string)
- modelo (string)
- ano (inteiro)
- cor (string)

A classe deve ter um método chamado "acelerar()" que imprime na tela a mensagem "Vrummm!".



Carro

EXERCÍCIOS POO – PARTE IV

Crie uma classe chamada "Cachorro". Essa classe deve ter os seguintes atributos:

- nome (string)
- raca (string)
- idade (inteiro)

A classe deve ter um método chamado "latir()" que imprime na tela a mensagem "Au au!".



Cachorro

EXERCÍCIOS POO – PARTE V



Pessoa

Crie uma classe chamada "Pessoa". Essa classe deve ter os seguintes atributos:

- nome (string)
- idade (inteiro)
- altura (float)

A classe deve ter um método chamado "apresentar()" que imprime na tela o nome e a idade da pessoa.

RESPOSTA III

```
class Carro {  
    public $marca;  
    public $modelo;  
    public $ano;  
    public $cor;  
  
    public function acelerar() {  
        echo "Vrummm!";  
    }  
}
```



RESPOSTA IV

```
class Cachorro {  
    public $nome;  
    public $raca;  
    public $idade;  
  
    public function latir() {  
        echo "Au au!";  
    }  
}
```



RESPOSTA V

```
class Pessoa {  
    public $nome;  
    public $idade;  
    public $altura;  
  
    public function apresentar() {  
        echo "Olá, meu nome é " . $this->nome . " e tenho " . $this->idade . " anos.";  
    }  
}
```

PILARES DA PROGRAMAÇÃO ORIENTADA A OBJETOS



A POO organiza o código em torno de objetos, facilitando a compreensão, manutenção e expansão de sistemas complexos.

- **Abstração:** Simplifica a representação de objetos, focando nas características essenciais;
- **Encapsulamento:** Protege os dados internos de um objeto e controla o acesso a eles;
- **Herança:** Cria novas classes a partir de classes existentes, promovendo a reutilização de código;
- **Polimorfismo:** Permite tratar objetos de diferentes classes de forma uniforme.

ABSTRAÇÃO

Abstração significa "esconder" partes da implementação do objeto expondo apenas uma interface simples para seu uso. Pense por exemplo num forno de micro-ondas, você não precisa entender toda a complexidade de como os componentes internos trabalham para gerar as ondas e produzir calor, você quer apenas apertar um ou dois botões e ter uma refeição quente pra comer. Toda essa complexidade são detalhes de implementação, você não precisa conhecê-los.

ENCAPSULAMENTO

Encapsulamento se refere à construção do objeto de modo a proteger o acesso direto a seus dados internos. Ao encapsularmos um objeto estamos agrupando propriedades e métodos que estão diretamente relacionados dentro de um mesmo objeto, permitindo que essas propriedades sejam acessadas apenas através de métodos públicos. Desta forma tratamos de questões importantes como segurança e confiabilidade do estado do objeto.

HERANÇA

A Herança é uma forma de eliminar repetição de código onde, como o próprio nome sugere, um objeto pode herdar características (ou seja, propriedades e métodos) de outra classe, sem a necessidade de se reescrever essas mesmas características.

POLIMORFISMO

Poli significa muitas e **Morphos** significa forma, então Polimorfismo significa muitas formas. Em POO Polimorfismo é caracterizado quando duas ou mais classes possuem métodos com o mesmo nome, mas podendo ter implementações diferentes. Assim, é possível utilizar qualquer objeto que implemente o mesmo método sem nos preocuparmos com o tipo do objeto. Na prática isso nos possibilita remover do nosso código diversos if ou switch cases.

CODIFICANDO ...



EXEMPLO DE UMA CALCULADORA

Imagine uma calculadora. Os botões e a tela são a interface pública (o que o usuário vê e interage). Internamente, a calculadora realiza cálculos complexos, mas esses detalhes são ocultos do usuário. O encapsulamento em POO funciona de forma similar:

- **Interface Pública:** Métodos que permitem ao usuário interagir com o objeto, por exemplo: `somar()`, `subtrair()`.

- **Detalhes Internos:** Atributos e lógica interna que realizam os cálculos, por exemplo, `$valor1`, `$valor2`, `$resultado`).

Calc.php

```
1  <?php
2  class Calc{
3      private $valor1;
4      private $valor2;
5      private $resultado;
6
7      public function somar($valor1,$valor2){
8          $resultado=$valor1+$valor2;
9          return $resultado;
10
11     }
12     public function subtrair($valor1,$valor2){
13         this->resultado=$valor1-$valor2;
14         return $resultado;
15
16     }
17     public function multiplicar($valor1,$valor2){
18         $resultado=$valor1*$valor2;
19         return $resultado;
20
21     }
```



```
22 public function dividir($valor1,$valor2){  
23     $resultado=$valor1/$valor2;  
24     return $resultado;  
25  
26 }  
27  
28 }  
29  
30 ?>
```

mostrar.php

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Calculadora</title>
5  </head>
6  <body>
7      <h1>Calculadora</h1>
8      <form method="post" action="">
9          <label for="numero1">Número 1:</label>
10         <input type="text" id="numero1" name="numero1" required><br><br>
11
12         <label for="numero2">Número 2:</label>
13         <input type="text" id="numero2" name="numero2" required><br><br>
14
15         <label for="operacao">Operação:</label>
16         <select id="operacao" name="operacao">
17             <option value="somar">Somar</option>
18             <option value="subtrair">Subtrair</option>
19             <option value="multiplicar">Multiplicar</option>
20             <option value="dividir">Dividir</option>
21         </select><br><br>
```

```
22
23     <input type="submit" value="Calcular">
24 </form>
25 </body>
26 </html>
27 <?php
28 if ($_SERVER["REQUEST_METHOD"] == "POST") {
29     require 'Calc.php';
30     $numero1 = $_POST['numero1'];
31     $numero2 = $_POST['numero2'];
32     $operacao = $_POST['operacao'];
33     $total=0;
34
35     $calc = new Calc();
36     if ($operacao=="somar"){
37         $total=$calc->somar($numero1,$numero2);
```

```
38
39     }
40     ✓ elseif($operacao=="subtrair"){
41         | $total=$calc->subtrair($numero1,$numero2);
42     }
43     ✓ elseif($operacao=="multiplicar"){
44         | $total=$calc->multiplicar($numero1,$numero2);
45     }
46     }
47     ✓ else{
48         | $total=$calc->dividir($numero1,$numero2);
49     }
50
51     echo "<h2>Resultado: $total</h2>";
52 }
53
54 ?>
```

EXPLICAÇÃO

Atributos Privados: `$valor1`, `$valor2` e `$resultado` são privados, ou seja, só podem ser acessados dentro da classe. Isso protege os dados internos de alterações acidentais.

Métodos Públicos:

- `setValor1()` e `setValor2()`: Permitem definir os valores dos operandos.
- `somar()` e `subtrair()`: Realizam as operações e armazenam o resultado em `$resultado`.
- `getResultado()`: Retorna o resultado do último cálculo.

Encapsulamento em Ação:

- O usuário da classe não precisa se preocupar com como a soma ou subtração são realizadas internamente. Ele apenas define os valores e chama os métodos necessários.
- A classe protege seus dados internos, evitando que sejam modificados de forma incorreta.

Modificadores de acesso

Com os modificadores de acesso determinamos a **visibilidade** de um método ou atributo pertencente a uma classe. Ou seja, definimos se ele pode ou não ser acessado fora da classe em que foi declarado.

Modificadores de acesso

Para modificar a visibilidade de um atributo ou método devemos preceder sua declaração de uma das palavras reservadas que representam o modificador, da seguinte forma:

```
1 | modificador $atributo;  
2 |  
3 | modificador function metodo() { }
```

Tipos

public

Este é o nível de acesso mais permissivo. Ele indica que o método ou atributo da classe é público, ou seja, pode ser acessado em qualquer outro ponto do código e por outras classes.

No código abaixo podemos ver um exemplo de uso do modificador de acesso public:

```
1 | class Exemplo {  
2 |     public $publico = 'Public';  
3 |     public function metodoPublico() { }  
4 | }
```

public

Este é o nível de acesso mais permissivo. Ele indica que o método ou atributo da classe é público, ou seja, pode ser acessado em qualquer outro ponto do código e por outras classes.

No código abaixo podemos ver um exemplo de uso do modificador de acesso public:

```
1 | class Exemplo {  
2 |     public $publico = 'Public';  
3 |     public function metodoPublico() { }  
4 | }
```

public

No código acima foi declarada uma classe denominada Exemplo que possui um atributo público chamado \$publico e um método público intitulado metodoPublico(). Neste caso ambos podem ser acessados por qualquer parte do código, incluindo outras classes.

private

Este modificador é o mais restrito. Com ele definimos que somente a própria classe em que um atributo ou método foi declarado pode acessá-lo. Ou seja, nenhuma outra parte do código, nem mesmo as classes filhas, pode acessar esse atributo ou método.

No código abaixo podemos ver um exemplo de uso do modificador de acesso private:

```
1 class Exemplo{  
2     private $privado = 'Privado';  
3     private function metodoPrivado() {}  
4 }
```


protected

Esse modificador indica que somente a própria classe e as classes que herdam dela podem acessar o atributo ou método. Dessa forma, ao instanciar a classe os elementos protegidos (protected) não podem ser acessados diretamente, como ocorre com o public.

A seguir podemos ver um exemplo de uso do modificador de acesso protected:

```
1 class Exemplo {  
2     protected $protegido = 'Protegido';  
3     protected function metodoProtegido() { }  
4 }
```

protected

No código acima foi declarada uma classe denominada Exemplo que possui um atributo protegido chamado \$protegido e um método protegido intitulado metodoProtegido(). Neste caso ambos só podem ser acessados pela própria classe e pelas suas classes filhas.

Exemplo – Conta Bancária

Crie uma pasta com o nome de: **conta_bancaria**

Crie o arquivo: **conta_bancaria.php** e com os seguintes comandos:

```
ContaBancaria.php
1  <?php
2  // Inicia a sessão para armazenar o saldo entre requisições
3  session_start();
4
5  // Definição da classe ContaBancaria
6  class ContaBancaria {
7      // Propriedade privada que armazena o saldo
8      // Variável privada, só está visível para essa classe
9      private $saldo;
10
```

```
10
11 // Construtor para inicializar o saldo com um valor da sessão ou padrão (zero)
12 ✓ public function __construct() {
13     // Se existir um saldo na sessão, utiliza-o, caso contrário, inicializa com 0
14     $this->saldo = isset($_SESSION['saldo']) ? $_SESSION['saldo'] : 0;
15 }
16
17 // Método set para definir o saldo
18 ✓ public function setSaldo($saldo) {
19     $this->saldo = $saldo;
20     // Atualiza o saldo na sessão para manter a persistência
21     $_SESSION['saldo'] = $this->saldo;
22 }
23
24 // Método get para obter o saldo atual
25 ✓ public function getSaldo() {
26     return $this->saldo;
27 }
```

```
28
29 // Método para depositar um valor na conta
30 public function depositar($quantia) {
31     // Verifica se a quantia é maior que zero antes de adicionar ao saldo
32     if($quantia > 0) {
33         //Pega o valor do saldo atual através da variável da classe (this->saldo) e soma
34         //E atualiza o setSaldo()
35         $saldoNovo = $this->saldo + $quantia;
36         $this->setSaldo($saldoNovo);
37     }
38 }
39
40 // Método para sacar um valor da conta
41 public function sacar($quantia) {
42     // Verifica se a quantia é válida e se há saldo suficiente
43     if($quantia > 0 && $quantia <= $this->getSaldo()) {
44         $this->setSaldo($this->getSaldo() - $quantia);
45     } else {
```

```
46         echo "Saldo insuficiente para o saque.<br>";
47     }
48 }
49 }
50 ?>
51
```


Exemplo – Conta Bancária

Crie um novo arquivo: **index.php** e com os seguintes comandos:

```
index.php
1  C:\Users\edwar\Desktop\USBWebserver v10\root\conta_bancaria\index.php
2  <html lang="pt-PT">
3  <head>
4      <meta charset="UTF-8">
5      <title>Exemplo de Conta Bancária com Set e Get</title>
6  </head>
7  <body>
8      <h2>Formulário da Conta Bancária</h2>
9
10     <!-- Formulário que permite ao usuário depositar ou sacar dinheiro -->
11     <form method="POST" action="">
12         Depósito: <input type="number" name="deposito" step="0.01" placeholder="Valor a depositar"><br><br>
13         Sacar: <input type="number" name="sacar" step="0.01" placeholder="Valor a sacar"><br><br>
14         <input type="submit" value="Enviar">
15     </form>
16
```

```
17 <?php
18 // Inclui o ficheiro que contém a classe ContaBancaria
19 require_once 'ContaBancaria.php';
20
21 // Cria ou recupera a instância da conta bancária
22 $conta = new ContaBancaria();
23
24 // Verifica se o formulário foi submetido
25 ✓ if ($_SERVER["REQUEST_METHOD"] == "POST") {
26     // Verifica se foi informado um valor para depósito
27     ✓ if (!empty($_POST['deposito'])) {
28         // Deposita o valor informado na conta
29         $conta->depositar(floatval($_POST['deposito']));
30     }
31
32     // Verifica se foi informado um valor para saque
33     ✓ if (!empty($_POST['sacar'])) {
34         // Tenta sacar o valor informado da conta
```

```
35         $conta->sacar(floatval($_POST['sacar']));
36     }
37 }
38
39 // Exibe o saldo atualizado da conta
40 echo "<h3>Saldo Atual</h3>";
41 echo "O saldo atual da conta é: R$" . number_format($conta->getSaldo(), 2, ',', '.') . "<br>";
42 ?>
43 </body>
44 </html>
45
```

Método Construtor

Tipo especial de método que inicializa as variáveis do objeto, quando instanciado (inicializado).

Construtores não possuem um retorno e utilizam a instrução **__construct**. Para que um construtor seja chamado é necessário o uso da palavra-chave `new` seguida pelo nome da classe e a lista de parâmetros, caso necessite.

Característica do Método Construtor

Não Retorna Valor: O construtor não pode retornar valores. Seu objetivo é apenas preparar o objeto.

Sobrecarga de Construtores: PHP não suporta sobrecarga de métodos de forma tradicional, o que significa que não é possível ter mais de um construtor na mesma classe. Para contornar essa limitação, pode-se usar parâmetros opcionais no construtor.

Exemplo de Classe com Método Construtor

Vamos criar uma classe chamada Carro que representa um carro. O construtor será usado para definir a marca, o modelo e o ano do carro.

```
<?php
class Carro {
    public $marca;
    public $modelo;
    public $ano;

    // Método construtor
    public function __construct($marca, $modelo, $ano) {
        $this->marca = $marca;
        $this->modelo = $modelo;
        $this->ano = $ano;
    }
}
```



```
// Método para exibir as informações do carro
public function exibirInformacoes() {
    echo "Marca: " . $this->marca . "\n";
    echo "Modelo: " . $this->modelo . "\n";
    echo "Ano: " . $this->ano . "\n";
}
}

// Criando um objeto da classe Carro
$meuCarro = new Carro("Toyota", "Corolla", 2020);

// Exibindo as informações do carro
$meuCarro->exibirInformacoes();
?>
```

Explicação do Código

1. Definição da Classe Carro:

- a) A classe Carro possui três propriedades públicas: marca, modelo, e ano.
- b) O método construtor **__construct(\$marca, \$modelo, \$ano)** é definido para inicializar essas propriedades quando um objeto da classe Carro é criado.

Explicação do Código

2. Método `exibirInformacoes()`:

- a) Esse método exibe as informações do carro (marca, modelo e ano) utilizando as propriedades do objeto.

3. Instanciando o Objeto:

- a) O objeto `$meuCarro` é criado usando a palavra-chave `new` e o construtor é automaticamente chamado com os parâmetros "Toyota", "Corolla", e 2020.
- b) As propriedades do objeto `$meuCarro` são inicializadas com esses valores.

Polimorfismo

Polimorfismo é um princípio da programação orientada a objetos que permite que objetos de diferentes classes sejam tratados como objetos de uma classe comum. Isso é possível através da herança e da implementação de interfaces.

Polimorfismo

Conceitos Básicos

Herança: Permite que uma classe derive propriedades e métodos de outra classe. A classe derivada (ou filha) herda os atributos e métodos da classe base (ou pai), podendo também sobrescrever métodos para fornecer implementações específicas.

Polimorfismo

Conceitos Básicos

Interfaces: São contratos que definem um conjunto de métodos que uma classe deve implementar. Elas permitem que diferentes classes implementem os mesmos métodos, garantindo que possam ser usadas de forma intercambiável.

Sobrecarga

Múltiplas funções com mesmo nome mas parâmetros diferentes são chamadas de sobrecarregadas.

Métodos podem ser sobrecarregados por uma mudança no número de argumentos e/ou nos tipos dos argumentos.

Muito útil para criarmos métodos semelhantes, mas que mudam de funcionamento de acordo com os parâmetros.

Exemplo

Múltiplas funções com mesmo nome mas parâmetros diferentes são chamadas de sobrecarregadas.

Métodos podem ser sobrecarregados por uma mudança no número de argumentos e/ou nos tipos dos argumentos.

Muito útil para criarmos métodos semelhantes, mas que mudam de funcionamento de acordo com os parâmetros.

Exemplo: Conta Bancaria

Arquivo: ContaBancaria.php

```
// Método para sacar um valor da conta
public function sacar($quantia) {
    // Verifica se a quantia é válida e se há saldo suficiente
    if($quantia > 0 && $quantia <= $this->getSaldo()) {
        $this->setSaldo($this->getSaldo() - $quantia);
    } else {
        echo "Saldo insuficiente para o saque.<br>";
    }
}
```

Exemplo: Conta Bancaria

Arquivo: ContaCorrente.php

```
// Sobrescreve o método sacar para considerar o limite da conta
// Polimorfismo
public function sacar($quantia) {
    // Calcula o saldo total disponível, incluindo o limite
    $saldoDisponivel = $this->getSaldo() + $this->limite;
```

Abstract - Abstração

Uma classe abstrata é uma classe que não pode ser instanciada diretamente. Ela serve como um modelo para outras classes, definindo métodos que devem ser implementados pelas classes derivadas.

Importância: As classes abstratas permitem criar uma estrutura comum para um grupo de classes relacionadas, garantindo que todas sigam um contrato específico.

Abstract - Abstração

Uma classe abstrata é uma classe que não pode ser instanciada diretamente. Ela serve como um modelo para outras classes, definindo métodos que devem ser implementados pelas classes derivadas.

Importância: As classes abstratas permitem criar uma estrutura comum para um grupo de classes relacionadas, garantindo que todas sigam um contrato específico.

Exemplo abstrato

Plano do Carro: Antes de construir um carro, é necessário ter um plano. Este plano diz que o carro terá **rodas**, **portas**, um **volante**, etc. Mas o plano por si só não é um carro, é apenas uma ideia de como o carro deve ser. Em programação, este "plano" é como uma **classe abstrata**.

Exemplo abstrato

Classe Abstrata em PHP: No PHP, uma "classe abstrata" funciona como esse plano. Ela define algumas **regras** ou **características** que outras classes devem seguir, mas por si só, não pode ser usada diretamente. É como dizer: "Todos os carros devem ter rodas, um motor e portas", mas sem construir um carro específico ainda.

Exemplo abstrato

Construir o Carro de Verdade: Agora, quando alguém quiser construir um carro de verdade, ele segue o plano (a classe abstrata) e cria um carro específico, como um carro de corrida ou um carro de passeio. Em PHP, isso significa que uma outra classe vai "**estender**" ou seguir o plano da classe abstrata, criando um "objeto" de verdade que pode ser usado.

Método abstrato

Um **método abstrato** é um tipo especial de método em programação orientada a objetos (OOP) que é definido em uma **classe abstrata** mas **não tem implementação** — ou seja, não tem o código que diz o que o método faz. Em vez disso, ele serve como um "esqueleto" ou "contrato" que define a assinatura do método (o nome, os parâmetros e o tipo de retorno), deixando a implementação concreta para as subclasses que herdam essa classe abstrata.

Método abstrato e classe abstrata

Classe Abstrata: Uma classe que não pode ser instanciada diretamente e serve como um modelo para outras classes.

Método Abstrato: Um método declarado em uma classe abstrata que obriga todas as subclasses a implementarem este método. É como dizer: "Quem herdar desta classe deve ter este método e decidir como ele funciona."

Exemplo prático

Crie uma pasta com o nome **Carro**, dentro da pasta **htdocs**;
Dentro da pasta crie a classe: **Veiculo.php**;

Veiculo.php

```
1  <?php
2  // Define uma classe abstrata chamada Veiculo.
3  // "abstract" indica que esta classe não pode ser instanciada diretamente.
4  // Serve como um plano que outras classes concretas devem seguir.
5  ▼ abstract class Veiculo {
6      // Declara duas propriedades protegidas (protected) da classe: marca e modelo.
7      // "protected" significa que essas propriedades podem ser acessadas na própria classe
8      // e nas classes que a estendem, mas não fora delas.
9      protected $marca;
10     protected $modelo;
11
12     // Método construtor da classe, chamado automaticamente quando um objeto é criado.
13     // Este método recebe dois parâmetros: marca e modelo.
14     // Define as propriedades $marca e $modelo da classe com os valores passados.
15     ▼ public function __construct($marca, $modelo) {
16         $this->marca = $marca; // Atribui o valor de $marca ao atributo $marca do objeto.
17         $this->modelo = $modelo; // Atribui o valor de $modelo ao atributo $modelo do objeto.
18     }
19
```

```
19
20 // Declaração de um método abstrato chamado mover().
21 // Um método abstrato não tem implementação aqui; as subclasses que herdam desta classe
22 // são obrigadas a definir esse método.
23 abstract public function mover();
24
25 // Método comum a todos os veículos para mostrar os detalhes (marca e modelo).
26 // Este método é "público", ou seja, pode ser chamado de fora da classe.
27 ✓ public function mostrarDetalhes() {
28     // Retorna uma string que descreve a marca e o modelo do veículo.
29     return "Marca: $this->marca, Modelo: $this->modelo";
30 }
31 }
32 ?>
33
```

Exemplo prático

Crie a classe: Carro.php;

```
Carro.php
1  <?php
2  // Inclui o arquivo 'Veiculo.php' uma vez, garantindo que ele só seja carregado uma vez.
3  // 'Veiculo.php' contém a definição da classe abstrata 'Veiculo' que será usada aqui.
4  require_once 'Veiculo.php';
5
6  // Define uma classe chamada 'Carro' que estende (herda) a classe abstrata 'Veiculo'.
7  // Isso significa que 'Carro' herda todas as propriedades e métodos definidos em 'Veiculo'.
8  class Carro extends Veiculo {
9
10     // Implementa o método abstrato 'mover()' que foi definido na classe 'Veiculo'.
11     // Todas as classes que herdam de uma classe abstrata precisam definir os seus próprios
12     // comportamentos para os métodos abstratos.
13     public function mover() {
14         // Retorna uma mensagem indicando que o carro está em movimento.
15         // A tag <br> é usada para inserir uma quebra de linha na saída HTML.
16         return "<br>O carro está andando na estrada!<br>";
17     }
18 }
19
20 ?>
```



Exemplo prático

Crie a classe: **Moto.php**;

```
Moto.php
1  <?php
2  // Inclui o arquivo 'Veiculo.php' uma vez, garantindo que ele só seja carregado uma vez.
3  // Este arquivo contém a definição da classe abstrata 'Veiculo'.
4  require_once 'Veiculo.php';
5  // Define uma classe chamada 'Mota' que estende (herda) a classe abstrata 'Veiculo'.
6  // Ao herdar de 'Veiculo', a classe 'Mota' também terá as propriedades e métodos da
7  // classe 'Veiculo'.
8  class Mota extends Veiculo {
9      // Implementa o método abstrato 'mover()' que foi definido na classe 'Veiculo'.
10     // Todas as classes que herdam de uma classe abstrata precisam fornecer uma
11     // implementação
12     // para os métodos abstratos declarados.
13     public function mover() {
14         // Retorna uma mensagem indicando que a mota está em movimento.
15         // A tag <br> é usada para inserir uma quebra de linha na saída HTML.
16         return "<br>A mota está a andar na estrada! <br>";
17     }
18 }
19
20 ?>
```

Exemplo prático

Crie a classe: index.php;

 index.php

```
1  <?php
2  // Inclui o arquivo 'Carro.php', que contém a definição da classe 'Carro'.
3  // 'require' garante que o script será interrompido com um erro fatal se o arquivo
4  não puder ser incluído.
5  require 'Carro.php';
6
7  // Inclui o arquivo 'Moto.php', que contém a definição da classe 'Mota'.
8  // Assim como em 'Carro.php', o uso de 'require' assegura que o script falhe se o
9  arquivo não estiver disponível.
10 require 'Moto.php';
```

```
11
12 // Cria uma nova instância da classe 'Carro'.
13 ✓ // O construtor da classe 'Carro' recebe dois parâmetros: "Toyota" como marca e "Corolla"
14 // como modelo.
15 $meuCarro = new Carro("Toyota", "Corolla");
16
17 // Chama o método 'mostrarDetalhes()' da instância '$meuCarro'.
18 // Este método, herdado da classe 'Veiculo', retorna uma string com a marca e o modelo
19 // do carro.
20 // A saída será: "Marca: Toyota, Modelo: Corolla"
21 echo $meuCarro->mostrarDetalhes();
22
23 // Chama o método 'mover()' da instância '$meuCarro'.
24 // Este método foi implementado especificamente na classe 'Carro' e retorna: "O carro
25 // está andando na estrada!<br>"
26 echo $meuCarro->mover();
```

```
27
28 // Cria uma nova instância da classe 'Mota'.
29 // O construtor da classe 'Mota' recebe dois parâmetros: "Honda" como marca e "CBR"
30 // como modelo.
31 $minhaMota = new Mota("Honda", "CBR");
32
33 // Chama o método 'mostrarDetalhes()' da instância '$minhaMota'.
34 // Assim como no caso do carro, este método retorna uma string com a marca e o
35 // modelo da moto.
36 // A saída será: "Marca: Honda, Modelo: CBR"
37 echo $minhaMota->mostrarDetalhes();
38
39 // Chama o método 'mover()' da instância '$minhaMota'.
40 // Este método foi implementado especificamente na classe 'Mota' e retorna: "A moto
41 // está a andar na estrada! <br>"
42 echo $minhaMota->mover();
43 ?>
```