

## Exercício 1 — Classe Abstrata Base

Crie uma **classe abstrata** chamada **Pessoa** que servirá de base para outras classes.

### Estrutura esperada:

```
<?php
abstract class Pessoa {
    protected $nome;
    protected $idade;
    protected $sexo;

    public function __construct($nome, $idade, $sexo) {
        $this->nome = $nome;
        $this->idade = $idade;
        $this->sexo = $sexo;
    }

    // Método comum (não abstrato)
    final public function fazerAniversario() {
        $this->idade++;
        echo "<p>Parabéns, {$this->nome}! Agora você tem
{$this->idade} anos.</p>";
    }

    // Método abstrato
    abstract public function apresentar();
}
```

### Tarefas:

1. Crie a classe acima.
2. Explique por que não é possível instanciá-la diretamente (`new Pessoa()` deve gerar erro). Porque ela é abstrata. Só serve para ser estendida por outras classes.
3. Identifique o papel do método **final**. Ele funciona do mesmo jeito em todo o código. Não pode ser mudado nas outras classes.
4. Explique a função de um método **abstrato**. É um método que não tem corpo. Ele obriga as outras classes a dizerem como ele funciona.

## Exercício 2 — Herança de Implementação

Crie uma classe chamada `Visitante` que herda de `Pessoa` e implementa o método abstrato `apresentar()`.

### Estrutura esperada:

```
class Visitante extends Pessoa {
    public function apresentar() {
        echo "<p>Sou um visitante chamado {$this->nome}.</p>";
    }
}
```

### Tarefas:

1. Implemente `Visitante`.
2. Instancie um visitante e teste os métodos herdados (`fazerAniversario()` e `apresentar()`).
3. Confirme que `Visitante` é uma **classe concreta** (instanciável). Ele é. Da pra criar objetos normalmente.
4. Essa herança é **pobre** ou **por diferença**? Justifique.

Ela é pobre porque ela só copia o básico da Pessoa, sem adicionar nada novo.

## Exercício 3 — Herança por Diferença

Crie uma classe `Aluno` que também herda de `Pessoa`, mas acrescenta novos atributos e métodos.

### Estrutura esperada:

```
class Aluno extends Pessoa {
    protected $matricula;
    protected $curso;

    public function __construct($nome, $idade, $sexo, $matricula,
    $curso) {
        parent::__construct($nome, $idade, $sexo);
    }
}
```

```

        $this->matricula = $matricula;
        $this->curso = $curso;
    }

    public function apresentar() {
        echo "<p>Sou o aluno {$this->nome}, do curso de
{$this->curso}</p>";
    }

    public function pagarMensalidade() {
        echo "<p>Mensalidade de {$this->nome} paga com
sucesso!</p>";
    }
}

```

### Tarefas:

1. Implemente `Aluno` conforme o modelo.
2. Explique o uso de `parent::__construct()`. Isso serve pra não repetir o código ao chamar o construct da classe pessoa.
3. Teste a criação de um objeto e verifique o método `fazerAniversario()` herdado.
4. Por que `fazerAniversario()` não pode ser sobrescrito? Porque ele tem o método 'final'.

## Exercício 4 — Subclasse com Sobrescrita e Novo Método

Crie uma classe `Bolsista` que **herda de** `Aluno`, adicionando um atributo e sobrescrevendo um método.

### Estrutura esperada:

```

class Bolsista extends Aluno {
    private $bolsa;

    public function __construct($nome, $idade, $sexo, $matricula,
$curso, $bolsa) {

```

```

        parent::__construct($nome, $idade, $sexo, $matricula,
$curso);
        $this->bolsa = $bolsa;
    }

    public function renovarBolsa() {
        echo "<p>Bolsa renovada para {$this->nome}!</p>";
    }

    public function pagarMensalidade() {
        echo "<p>{$this->nome} é bolsista! Pagamento com desconto de
{$this->bolsa}%.</p>";
    }
}

```

### Tarefas:

1. Implemente **Bolsista** e teste os métodos.
2. Identifique o conceito de **polimorfismo** no método **pagarMensalidade()**. É quando um mesmo método funciona de jeitos diferentes dependendo da classe.
3. Analise: o método **fazerAniversario()** pode ser sobrescrito? Por quê? Não pode porque o código possui o método 'final'.

## Exercício 5 — Classe Final e Hierarquia Completa

Crie uma classe **Professor** que herda de **Pessoa**, e declare-a como **final**, impedindo novas heranças.

### Estrutura esperada:

```

final class Professor extends Pessoa {
    private $especialidade;
    private $salario;

    public function __construct($nome, $idade, $sexo, $esp,
$salario) {
        parent::__construct($nome, $idade, $sexo);
        $this->especialidade = $esp;
    }
}

```

```
        $this->salario = $salario;
    }

    public function apresentar() {
        echo "<p>Sou o professor {$this->nome}, especialista em
{$this->especialidade}</p>";
    }

    public function receberAumento($valor) {
        $this->salario += $valor;
        echo "<p>O salário de {$this->nome} foi reajustado para R$
{$this->salario}</p>";
    }
}
```

### Tarefas:

1. Crie a classe `Professor` com `final`.
2. Tente criar uma classe `Coordenador` `extends Professor` e observe o erro.
3. Crie um vetor com um `Visitante`, um `Aluno`, um `Bolsista` e um `Professor`.
4. Use `get_class($obj)` e `instanceof` para identificar a hierarquia.
5. Identifique qual é a **raiz** e quais são as **folhas** da árvore de herança. A raiz é a classe 'pessoa' e as folhas são 'visitante', 'bolsista' e 'professor'.