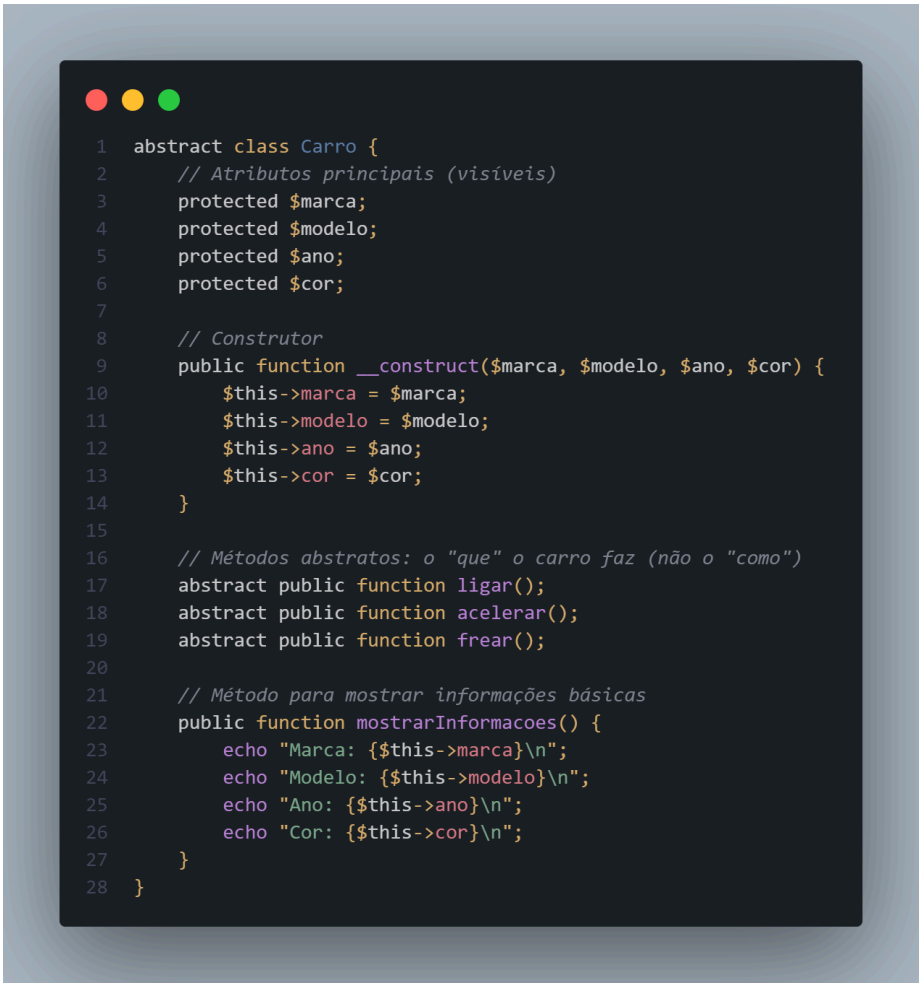
Teste Teórico - Merchion
Cauê Marchi Foyth

1. Orientação a Objetos (OOP)
   a) Explique os quatro pilares da Orientação a Objetos. Dê exemplos práticos de cada pilar (mesmo que breves) para demonstrar entendimento.
   R: A Orientação a Objetos é composta por 4 pilares, dentre eles: Abstração, Encapsulamento, Herança e Polimorfismo.

Abstração: O objetivo dele é “mostrar só o que importa”, ou seja é o processo de ocultar detalhes complexos e mostrar apenas as características principais, concentrando no que o objeto faz, em vez de como ele faz. Por exemplo: Uma classe “Carro”, pode ter atributos como “marca”, “modelo”, “ano” e “cor”, e métodos como “ligar”, “acelerar” e “frear”, com a abstração não precisa detalhar como o carro liga ou como acelera.
Exemplo em código:



```php
abstract class Carro {
    // Atributos principais (visíveis)
    protected $marca;
    protected $modelo;
    protected $ano;
    protected $cor;

    // Construtor
    public function __construct($marca, $modelo, $ano, $cor) {
        $this->marca = $marca;
        $this->modelo = $modelo;
        $this->ano = $ano;
        $this->cor = $cor;
    }

    // Métodos abstratos: o "que" o carro faz (não o "como")
    abstract public function ligar();
    abstract public function acelerar();
    abstract public function frear();

    // Método para mostrar informações básicas
    public function mostrarInformacoes() {
        echo "Marca: {$this->marca}\n";
        echo "Modelo: {$this->modelo}\n";
        echo "Ano: {$this->ano}\n";
        echo "Cor: {$this->cor}\n";
    }
}
```

Exemplo de implementação:

```php
class CarroEsportivo extends Carro {

    public function ligar() {
        echo "O carro esportivo foi ligado com botão Start/Stop.\n";
    }

    public function acelerar() {
        echo "O carro está acelerando rapidamente!\n";
    }

    public function frear() {
        echo "O carro está freando com freios ABS.\n";
    }
}

// Utilização
$meuCarro = new CarroEsportivo("Ferrari", "488 GTB", 2020, "Vermelho");

$meuCarro->mostrarInformacoes();
$meuCarro->ligar();
$meuCarro->acelerar();
$meuCarro->frear();
```

Encapsulamento: Tem como objetivo proteger os dados e métodos de um objeto, usando modificadores de acesso. Resumidamente “Guardar as coisas dentro”, trazendo para um exemplo prático, vamos pensar em um “Caixa de brinquedos”, todos os brinquedos estão guardados lá dentro, porém ninguém precisa saber todos os brinquedos que tem dentro, só precisa saber que é uma caixa de brinquedos.
Exemplo em código:

```php
class CaixaDeBrinquedos {
    // Atributo privado: ninguém de fora pode acessar diretamente
    private $brinquedos = [];

    // Atributo protegido: só a classe e seus filhos podem acessar
    protected $dono;

    // Atributo público: pode ser acessado de fora
    public $descricao;

    // Construtor
    public function __construct($descricao, $dono) {
        $this->descricao = $descricao;
        $this->dono = $dono;
    }

    // Método público: qualquer um pode usar para adicionar brinquedos
    public function adicionarBrinquedo($brinquedo) {
        $this->brinquedos[] = $brinquedo;
        echo "Brinquedo '$brinquedo' adicionado à caixa.\n";
    }

    // Método público: mostra quantos brinquedos tem na caixa
    public function contarBrinquedos() {
        return count($this->brinquedos);
    }

    // Método privado: usado apenas internamente pela classe
    private function listarBrinquedos() {
        return implode(', ', $this->brinquedos);
    }

    // Método público para mostrar os brinquedos (de forma controlada)
    public function mostrarBrinquedos() {
        echo "Brinquedos na caixa: " . $this->listarBrinquedos() . "\n";
    }
}
```

Exemplo de implementação:

```php
$caixa = new CaixaDeBrinquedos("Caixa colorida com tampa", "Maria");

// Usando métodos públicos para interagir com a caixa
$caixa->adicionarBrinquedo("Carrinho");
$caixa->adicionarBrinquedo("Boneca");

echo "Descrição da caixa: {$caixa->descricao}\n";
echo "Total de brinquedos: " . $caixa->contarBrinquedos() . "\n";

$caixa->mostrarBrinquedos();

// Abaixo estão exemplos de tentativas inválidas (gerariam erro)

echo $caixa->brinquedos; // ERRO: $brinquedos é private
echo $caixa->dono;       // ERRO: $dono é protected
$caixa->listarBrinquedos(); // ERRO: método private

```

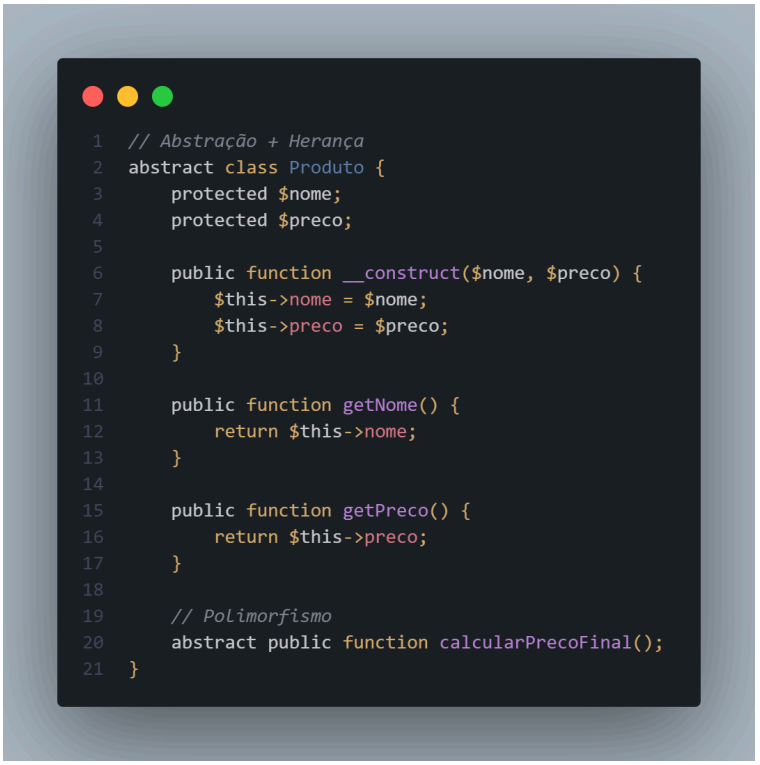
Herança: Permite que uma classe filha herde atributos e métodos de uma classe pai, assim permitindo uma reutilização de código, extensibilidade e organização hierárquica, geralmente aplicando um conceito de “Generalização para especialização”, onde classes mais genéricas são estendidas para classes mais específicas. Trazendo para um exemplo, uma classe “Veículo” pode ser especializada em uma classe “Carro” e “Moto”, herdando atributos e métodos mais genéricos.
Exemplo em código:

```php
class Veiculo {
    protected $marca;
    protected $modelo;

    public function __construct($marca, $modelo) {
        $this->marca = $marca;
        $this->modelo = $modelo;
    }

    public function ligar() {
        echo "O veículo {$this->marca} {$this->modelo} está ligado.\n";
    }

    public function desligar() {
        echo "O veículo {$this->marca} {$this->modelo} está desligado.\n";
    }

    public function mostrarInformacoes() {
        echo "Marca: {$this->marca}\n";
        echo "Modelo: {$this->modelo}\n";
    }
}

// Classe Carro que herda de Veiculo
class Carro extends Veiculo {
    private $portas;

    public function __construct($marca, $modelo, $portas) {
        parent::__construct($marca, $modelo); // chama o construtor da classe pai
        $this->portas = $portas;
    }

    public function mostrarInformacoes() {
        parent::mostrarInformacoes(); // reutiliza o método da classe pai
        echo "Portas: {$this->portas}\n";
    }

    public function buzinar() {
        echo "Carro buzinando: Bibi!\n";
    }
}

// Classe Moto que herda de Veiculo
class Moto extends Veiculo {
    private $cilindradas;

    public function __construct($marca, $modelo, $cilindradas) {
        parent::__construct($marca, $modelo);
        $this->cilindradas = $cilindradas;
    }

    public function mostrarInformacoes() {
        parent::mostrarInformacoes();
        echo "Cilindradas: {$this->cilindradas}cc\n";
    }

    public function empinar() {
        echo "A moto está empinando!\n";
    }
}
```

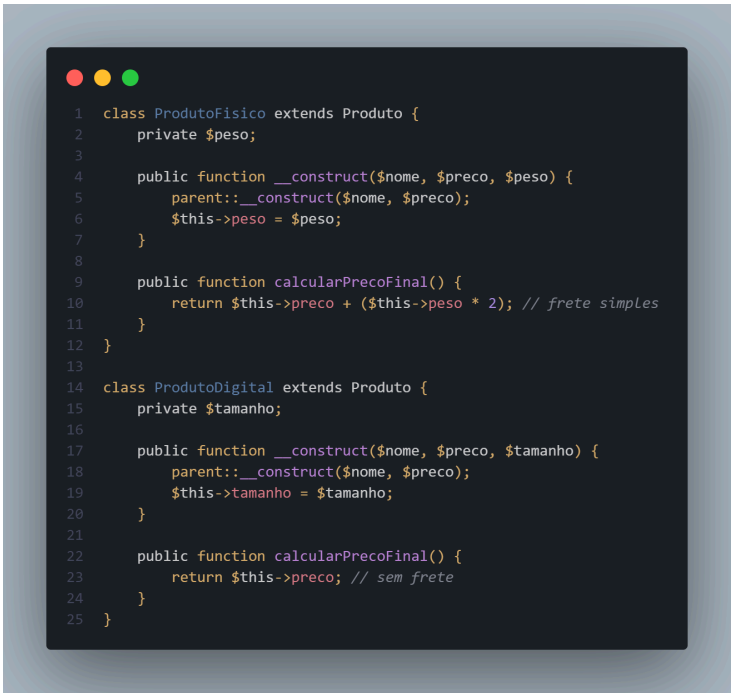
Polimorfismo: É a capacidade de diferentes classes responderem de forma diferente a uma mesma chamada de método, geralmente quando essas classes herdam de uma mesma superclasse. Isso permite que objetos diferentes sejam tratados de forma uniforme, mas cada um tenha um comportamento específico. Por exemplo, imagine uma classe “Animal” com um método “fazerSom”. Classes como “Cachorro” e “Gato” podem herdar de Animal, mas cada uma implementa “fazerSom” de forma diferente, o cachorro late e o gato mia.
Exemplo em código:

```php
abstract class Animal {
    protected $nome;

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

    // Método abstrato (obrigatório nas subclasses)
    abstract public function fazerSom();
}

// Subclasse: Cachorro
class Cachorro extends Animal {
    public function fazerSom() {
        echo "{$this->nome} diz: Au au!\n";
    }
}

// Subclasse: Gato
class Gato extends Animal {
    public function fazerSom() {
        echo "{$this->nome} diz: Miau!\n";
    }
}

// Função que aceita qualquer Animal
function emitirSomDoAnimal(Animal $animal) {
    $animal->fazerSom();
}

// ----- Testando o polimorfismo -----

$animais = [
    new Cachorro("Rex"),
    new Gato("Mingau"),
];

foreach ($animais as $animal) {
    emitirSomDoAnimal($animal); // Cada animal reage com seu som específico
}

```

b) Em PHP, como você aplica esses princípios ao construir uma classe de domínio (por exemplo, Produto ou Pedido)?
R: Aplicaria os 4 princípios da Orientação a Objetos, garantindo um código limpo, reutilizável, seguro, e fácil de manter.
Exemplo:
Abstração: Classe abstrata Produto, com método abstrato calcularPrecoFinal()
Encapsulamento: Atributos privados e protegidos, controle de acesso por métodos públicos
Herança: ProdutoFisico e ProdutoDigital herdam de Produto
Polimorfismo: Método calcularPrecoFinal() chamado de forma uniforme no Pedido



```php
// Abstração + Herança
abstract class Produto {
    protected $nome;
    protected $preco;

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

    public function getNome() {
        return $this->nome;
    }

    public function getPreco() {
        return $this->preco;
    }

    // Polimorfismo
    abstract public function calcularPrecoFinal();
}
```



```php
class ProdutoFisico extends Produto {
    private $peso;

    public function __construct($nome, $preco, $peso) {
        parent::__construct($nome, $preco);
        $this->peso = $peso;
    }

    public function calcularPrecoFinal() {
        return $this->preco + ($this->peso * 2); // frete simples
    }
}

class ProdutoDigital extends Produto {
    private $tamanho;

    public function __construct($nome, $preco, $tamanho) {
        parent::__construct($nome, $preco);
        $this->tamanho = $tamanho;
    }

    public function calcularPrecoFinal() {
        return $this->preco; // sem frete
    }
}
```

```php
class Pedido {
    private $produtos = [];

    public function adicionarProduto(Produto $produto) {
        $this->produtos[] = $produto;
    }

    public function calcularTotal() {
        $total = 0;
        foreach ($this->produtos as $p) {
            $total += $p->calcularPrecoFinal(); // polimorfismo
        }
        return $total;
    }

    public function listarProdutos() {
        foreach ($this->produtos as $p) {
            echo "- {$p->getNome()} (R$ " . number_format($p->calcularPrecoFinal(), 2) . ")\n";
        }
    }
}
```

Teste final:

```php
$pedido = new Pedido();

$pedido->adicionarProduto(new ProdutoFisico("Notebook", 3000, 2.5));
$pedido->adicionarProduto(new ProdutoDigital("Curso PHP", 199.90, 1500));

echo "Produtos no pedido:\n";
$pedido->listarProdutos();

echo "\nTotal do pedido: R$ " . number_format($pedido->calcularTotal(), 2) . "\n";
```

2. TypeScript
   a) Quais as vantagens de se usar TypeScript em projetos Vue (ou JavaScript em geral)?
   R: TypeScript traz diversas vantagens com a tipagem estática em comparação a utilização de javaScript, porém dentre as vantagens existem duas das quais vejo como ponto chave para a utilização de TypeScript em projetos. A primeira vantagem é a detecção precoce de erros, é possível identificar erros no desenvolvimento, antes mesmo da execução da aplicação, assim reduzindo a quantidade de erros. A segunda peça chave para a utilização é a escalabilidade que o TypeScript fornece em um projeto que passa na mão de diversos desenvolvedores, entregando uma maior consistência de código e uma redução na complexidade cognitiva, assim facilitando por exemplo a um novo desenvolvedor dar manutenção no código.

b) Explique a diferença entre tipagem estática e tipagem dinâmica no contexto de TypeScript e JavaScript.
R: Em tipagem estática no TypeScript, as variáveis são definidas no momento da escrita do código e verificada antes da execução, ou seja no momento de compilação. O tipo não muda depois de definido. Ajuda a evitar bugs, melhora o autocomplete e torna o código mais previsível.
Na tipagem dinâmica do JavaScript, as variáveis tem tipos definidos automaticamente com base no valor atribuído, podem mudar em tempo de execução. Mais flexível, porém mais propenso a erros que só aparece em tempo de execução, assim trazendo menos segurança.

3. PHP

a) Autoloading e Composer: O que é autoloading no PHP? Explique como o Composer facilita esse processo e por que esse recurso é fundamental em projetos de médio e grande porte.
R: Autoloading é o processo de carregar automaticamente as classes de um projeto PHP quando elas são usadas, sem precisar dar “require” manualmente para cada arquivo. Composer além de ser o gerenciador de dependências do PHP ele também gera automaticamente um autoloader baseado no padrão PSR-4, mapeando namespaces para diretórios. Ele cria um arquivo vendor/autoload.php que, registra uma função autoloader, mapeia namespaces para caminhos de arquivos e carrega classes sob demanda.

b) Tratamento de Exceções: Como você lida com exceções em PHP? Dê exemplos de como criar suas próprias exceções personalizadas e explique quando isso é benéfico.
R: O tratamento de exceções em PHP utiliza blocos de código com o “try”, “catch” e “finally”. Permite interromper o fluxo normal do código quando acontece algo fora do padrão, podendo capturar e tratá-lo de forma controlada, sem apresentar um log ou retorno feio ou até mesmo em casos mais críticos travar a aplicação.

c) PSRs (PHP Standards Recommendations): Cite duas ou três PSRs que você considera importantes no dia a dia de um projeto PHP e por que elas são relevantes.
R: PSR-4 Define como as classes devem ser carregadas automaticamente com base em seus namespaces e estrutura de diretórios, eliminando a necessidade de “requires” manualmente, facilitando a organização e manutenção do código.
PSR-12 Estabelece convenções de formatação e estilo de código, melhorando legibilidade e manutenção, facilita a colaboração em equipe, entre outras vantagens.
PSR-7 HTTP Message Interface, Define interfaces para representar mensagens HTTP, assim padroniza a manipulação de requests e response, facilita a testabilidade e promove imutabilidade e segurança.

4. Integração com Frameworks

a) Explique a diferença entre SSR (Server-Side Rendering), SSG (Static Site Generation) e SPA (Single Page Application).
R: No SSR, o servidor monta o HTML na hora de requisição, toda vez que essa página é acessada o conteúdo é gerado dinamicamente pelo servidor, suas vantagens é ser

bom para SEO e manter o conteúdo sempre atualizado, suas desvantagens é ser mais lento que o SSG e totalmente dependente da resposta do servidor.
No SSG o conteúdo é pré-renderizado no momento do build, os arquivos HTML são estáticos e servidos diretamente, tem como vantagem uma alta performance e ser barato de ser hospedado, em contrapartida sua desvantagem é que o conteúdo não muda facilmente em tempo real, assim limitando para conteúdos não muito dinâmicos.
Na SPA o navegador carrega um único HTML vazio, e a partir disso o conteúdo é montado via JavaScript no cliente, tem como vantagem a experiência fluida, pois não precisa carregar a página, assim sendo ideal para dashboards por exemplo, e tem como desvantagem sendo pior para SEO, pois o HTML inicial chega vazio, e pode ter um tempo inicial de carregamento mais alto.

b) Laravel: Explique detalhadamente o ciclo de vida de uma requisição HTTP no Laravel, desde o momento em que a requisição chega ao servidor até a resposta ser enviada ao cliente. Destaque os principais componentes do framework envolvidos nesse processo e explique o papel de cada um (por exemplo: middleware, roteamento, controllers, service providers, etc).
   R: A requisição começa chegando no “index.php”, onde é o ponto de entrada da aplicação e também onde o Laravel é iniciado, a partir disso o Laravel passa a requisição para o Kernel que é o coração do sistema, ele define e executa os middlewares que são responsável por filtros, autenticações, proteger contra ataques, e se estiver tudo de acordo eles deixam a requisição seguir, com isso ele deixa cair na rota certa, da qual redireciona para a Controller, aqui vai ser feita a lógica do sistema, buscando dados, chamando serviços, validações e entre outras, pode usar Models ou Services caso necessário, assim retornando uma resposta pode ser uma View, um JSON ou um redirect, assim a resposta sobe de volta para o Kernel e vai para o destino final.