

Questões Teóricas

1. Orientação a Objetos (OOP)

Os quatro pilares da Orientação a Objetos:

Encapsulamento:

- Descrição: Esconder os detalhes internos de implementação de um objeto, expondo apenas a interface que é necessária para interação, normalmente usado através de métodos (**getters/setters**).
- Exemplo: Em uma classe **Produto**, os atributos **\$nome** e **\$preço** são privados e só podem ser acessados por métodos públicos como **getNome** e **getPreço**.

```
<?php
no usages
class Produto {
    2 usages
    private string $nome;
    2 usages
    private float $preço;

    no usages
    public function __construct(string $nome, float $preço) {
        $this->nome = $nome;
        $this->preço = $preço;
    }

    no usages
    public function getNome(): string
    {
        return $this->nome;
    }

    no usages
    public function getPreço(): float
    {
        return $this->preço;
    }
}
```

Herança:

- Descrição: Permitir que uma nova classe herde atributos e métodos de uma classe existente.
- Exemplo: Uma classe **Funcionario** que herda de uma classe **Pessoa**.

```
class Pessoa {
    //TODO: Para o atributo ser acessível pela classe que está estendendo a classe Pessoa,
    // a visibilidade precisa ser protected (protegida)
    protected string $nome;
    private string $identidade;
    public function __construct(string $nome)
    {
        $this->nome = $nome;
    }
    protected function getNome(): string
    {
        return $this->nome;
    }
    private function getIdentidade(): string
    {
        return $this->identidade;
    }
}

class Funcionario extends Pessoa {
    private float $salario;
    public function __construct(string $nome, float $salario)
    {
        parent::__construct($nome);
        $this->salario = $salario;
    }
    public function getSalario(): float
    {
        //TODO: Observe que na classe Funcionario, não existe o método getNome, ele é herdado da classe Pessoa.
        $nomeFuncionario = $this->getNome();

        //TODO: O mesmo não acontece com o método getIdentidade, pois a visibilidade dele foi definida como private.
        $identidade = $this->getIdentidade();
        return $this->salario;
    }
}
```

codestrap.de

Polimorfismo:

- Descrição: Capacidade de um método assumir diferentes formas, permitindo que objetos de diferentes classes sejam tratados de maneira uniforme.
- Exemplo: Métodos **calcularSalario** diferentes em classes **Funcionario** e **Gerente**, ambos herdam de **Pessoa**.

```
class Pessoa
{
    public function calcularSalario(): float
    {
        return 0;
    }
}

class Funcionario extends Pessoa
{
    public function calcularSalario(): float
    {
        return 3000;
    }
}

class Gerente extends Funcionario
{
    public function calcularSalario(): float
    {
        return 5000;
    }
}

function exibirSalario(Pessoa $pessoa)
{
    //TODO: Aqui utilizamos o polimorfismo para chamar o método apropriado
    echo $pessoa->calcularSalario() . PHP_EOL;
}

$pessoa = new Pessoa();
$funcionario = new Funcionario();
$gerente = new Gerente();

exibirSalario($pessoa);
exibirSalario($funcionario);
exibirSalario($gerente);
```

codesnap.dev

Abstração:

- Descrição: Foca nos aspectos essenciais de um objeto, ignorando detalhes desnecessários.
- Exemplo: Uma classe abstrata **Veiculo** que define métodos gerais para veículos, sem implementar detalhes específicos.

```
abstract class Veiculo
{
    protected string $marca;
    protected string $modelo;
    public function __construct(string $marca, string $modelo)
    {
        $this->marca = $marca;
        $this->modelo = $modelo;
    }
    abstract public function andar();
}

class Carro extends Veiculo
{
    public function andar(): string
    {
        return "O carro está andando";
    }
}

class Moto extends Veiculo
{
    public function andar(): string
    {
        return "A Moto está andando";
    }
}

//TODO: Aqui cada veiculo tem uma marca e modelo diferente, mas os dois realizam o mesmo método geral de andar
$carro = new Carro('Audi', 'A3');
$carro->andar();

$moto = new Moto('Honda', 'Biz');
$moto->andar();
```

codesnap.dev

2. TypeScript

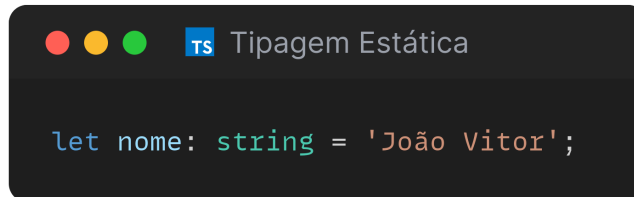
Vantagens de Usar TypeScript:

3. **Tipagem Estática:** Reduz erros em tempo de execução, detectando inconsistências durante a compilação.
4. **Autocompletar e IntelliSense:** Melhora a produtividade fornecendo sugestões de código.
5. **Refatoração Segura:** Facilita a modificação do código sem introduzir novos bugs.
6. **Documentação Implícita:** O código bem tipado serve como documentação, facilitando o entendimento por outros desenvolvedores.
7. **Suporte da Comunidade:** Ampla adoção e suporte contínuo da comunidade de desenvolvedores e da Microsoft.

Tipagem Estática vs. Tipagem Dinâmica

Tipagem Estática:

- **Definição:** As variáveis têm seus tipos definidos no tempo de compilação.
- **Exemplo:** Em TypeScript:



```
let nome: string = 'João Vitor';
```

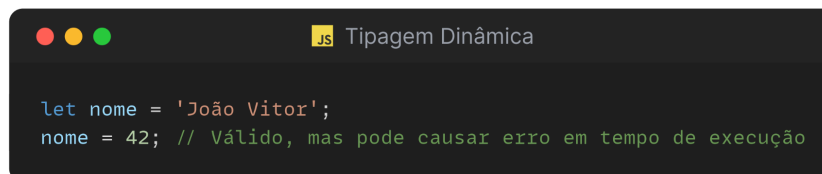
codesnap.dev

Vantagens:

- Detecção de erros antes da execução do código.
- Melhor suporte para ferramentas de desenvolvimento.

Tipagem Dinâmica:

- **Definição:** As variáveis têm seus tipos definidos no tempo de execução.
- **Exemplo:** Em JavaScript:



```
let nome = 'João Vitor';  
nome = 42; // Válido, mas pode causar erro em tempo de execução
```

codesnap.dev

Vantagens:

- Maior flexibilidade e rapidez no desenvolvimento inicial.
- Menor quantidade de código necessário para definir tipos.

O **TypeScript** oferece uma estrutura mais robusta para projetos de médio a grande porte, especialmente quando se busca manter e escalar o código a longo prazo. A tipagem estática ajuda a prevenir muitos erros comuns que podem passar despercebidos em linguagens com tipagem dinâmica.

3. PHP

Autoloading e Composer

Autoloading é uma funcionalidade em PHP que carrega automaticamente as classes necessárias sem a necessidade de incluí-las manualmente com **require** ou **include**. Isso facilita o desenvolvimento e a manutenção do código, pois evita a inclusão redundante e minimiza os erros de inclusão de arquivos.

Composer é uma ferramenta de gerenciamento de dependências em PHP que, além de gerenciar pacotes, simplifica o processo de autoloading através do padrão **PSR-4 (Autoloader)**. O padrão especifica que você deve seguir o seguinte formato para o namespace de um arquivo:

"\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>"

1. **Instalação de pacotes:** Composer permite a instalação de bibliotecas e dependências de forma simples com o comando **composer require**.
2. **Arquivo composer.json:** Mantém as dependências do projeto listadas.
3. **Autoloading:** Ao configurar corretamente o composer.json, o composer gera um arquivo autoload.php que segue o padrão **PSR-4**, facilitando o carregamento automático de classes.

Importância em Projetos de Médio e Grande Porte:

- **Organização:** Facilita a organização do código e modularização.
- **Escalabilidade:** Simplifica a manutenção e a adição de novas funcionalidades.
- **Produtividade:** Reduz o tempo gasto com inclusão manual de classes.

Tratamento de Exceções

O tratamento de exceções em PHP é feito utilizando **try**, **catch** e **finally**.

```
try {
    // Código que pode gerar uma exceção
    throw new Exception("Erro ocorrido durante a execução!");
} catch (Exception $e) {
    // Tratamento da exceção
    echo "Exception capturada: " . $e->getMessage();
} finally {
    // Código que sempre será executado
    return true;
}
```

codesnap.dev

Exceções Personalizadas: As exceções personalizadas são úteis para tratar erros específicos de uma sistema.

Exemplo:

```
class HandleDefinitionException extends Exception
{
    public function getMessageError(): string
    {
        return "Erro específico: {$this->message}";
    }
}

try {
    throw new HandleDefinitionException("Quebrou alguma coisa aqui");
} catch (HandleDefinitionException $e) {
    echo $e->getMessageError();
}
```

codesnap.dev

Benefícios:

- **Clareza:** Ajuda a diferenciar tipos de erros.
- **Manutenção:** Facilita o rastreamento de problemas específicos.
- **Modularidade:** Permite um tratamento mais detalhado e contextualizado.

PSRs (PHP Standards Recommendations)

Algumas PSRs importantes no dia a dia de um projeto PHP são:

1. **PSR-1 (Basic Coding Standard):** Define padrões básicos de codificação, como o uso de tags de PHP (`<?php ?>`) e normas para métodos e classes. É relevante porque garante consistência no código, facilitando a leitura e manutenção.
2. **PSR-4 (Autoloading Standard):** Define padrões para autoloading de classes, facilitando a organização do código e a integração de bibliotecas. É essencial para o autoloading eficiente e manutenção escalável de projetos.
3. **PSR-12 (Extended Coding Style Guide):** Expande as diretrizes do **PSR-1** com recomendações detalhadas sobre formatação de código, incluindo indentação, espaçamento e uso de declarações. Ajuda a manter um código limpo e padronizado, essencial para equipes de desenvolvimento.

4. Integração com Frameworks

Vue/Nuxt: SSR, SSG e SPA

- **SSR (Server-Side Rendering):**
 - **Definição:** A renderização ocorre no servidor e o HTML completo é enviado ao cliente.
 - **Vantagens:** Melhor desempenho em SEO, tempo de carregamento inicial rápido.
 - **Como o Nuxt Lida:** Suporta SSR nativamente, gerando páginas no servidor antes de enviar ao cliente.
- **SSG (Static Site Generation):**
 - **Definição:** As páginas são geradas estaticamente no momento da construção (build time) e servidas como arquivos HTML estáticos.
 - **Vantagens:** Desempenho rápido, ótimo para conteúdos que não mudam frequentemente.
 - **Como o Nuxt Lida:** Permite a geração de sites estáticos, criando arquivos HTML durante o processo de build.
- **SPA (Single Page Application):**
 - **Definição:** A aplicação é carregada como uma única página, e a navegação subsequente ocorre no cliente, sem recarregar a página.
 - **Vantagens:** Experiência de usuário mais fluida e rápida após o carregamento inicial.
 - **Como o Nuxt Lida:** Pode ser configurado para funcionar como uma SPA, mantendo a navegação sem recarregar a página.

Nuxt é um framework que permite a escolha entre essas abordagens conforme as necessidades do projeto, proporcionando flexibilidade e eficiência.

Symfony: Service Container e Injeção de Dependência

O **Service Container** no Symfony é um componente essencial para a **injeção de dependência**. Ele gerencia a criação e a injeção de serviços e suas dependências.

- **Service Container:**
 - **Definição:** É um contêiner que armazena e gerencia a instância dos serviços (objetos PHP) que a aplicação usa.
 - **Papel:** Facilita a injeção de dependência, criando e injetando automaticamente os serviços necessários nas classes, promovendo um código mais modular e fácil de manter.
- **Injeção de Dependência:**
 - **Definição:** É um padrão de design que permite que os objetos recebam suas dependências em vez de criá-las diretamente.
 - **Benefícios:** Facilita a testabilidade e o desacoplamento do código.

Exemplo de Configuração Simples:

Definindo um serviço:

```
namespace App\Service;

class MyService
{
    public function implementarLogica()
    {
        // Lógica do serviço
    }
}
```

codesnap.dev

Configurando o serviço no contêiner (não é necessário usar a convenção de autowiring):

```
services:
    App\Service\MyService:
        autowire: true
        autoconfigure: true
```

codesnap.dev

Injetando o serviço em um controller:

```
namespace App\Controller;

use App\Service\MyService;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class MyController extends AbstractController
{
    private MyService $myService;

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

    public function index(): Response
    {
        $this->myService->implementarLogica();
        return new Response('Serviço executado com sucesso!');
    }
}
```

codesnap.dev

Neste exemplo, o **Service Container** do Symfony cria uma instância de **MyService** e a injeta no **MyController**, permitindo que o serviço seja utilizado de forma simples e organizada.