ChatGPT

Revisão Completa: HTML, CSS, JavaScript e PHP (Do Iniciante ao Avançado)

Introdução: Este material fornece uma revisão abrangente das tecnologias **HTML, CSS, JavaScript e PHP**, organizadas por nível de dificuldade (iniciante, intermediário e avançado). Cada seção apresenta explicações teóricas claras, exemplos práticos comentados e exercícios com sugestões de solução. HTML e CSS tratam da estrutura e apresentação de páginas web no **lado do cliente**, enquanto JavaScript adiciona interatividade também no cliente. Já o PHP é uma linguagem executada no **lado do servidor**, responsável por lógica e geração dinâmica de conteúdo [1].

Vamos começar nossa revisão tecnológica, lembrando sempre das **boas práticas modernas** em cada área. Ao longo do texto, fornecemos tabelas comparativas e exemplos de código devidamente comentados para facilitar o entendimento. Bons estudos!

HTML

HTML Iniciante

Conceitos Básicos: HTML (HyperText Markup Language, em português *Linguagem de Marcação de Hipertexto*) é a base da estrutura de páginas web. Consiste em **elementos** marcados por *tags* que definem o significado do conteúdo (por exemplo, um parágrafo, um título, uma imagem). Um arquivo HTML típico é um documento de texto com extensão `.html` que o navegador interpreta e renderiza visualmente.

Estrutura de um Documento HTML: Todo documento HTML segue uma estrutura básica composta pelos elementos `<html>`, `<head>` e `<body>`. No topo, é **obrigatório declarar o tipo do documento** usando `<!DOCTYPE html>`, que indica ao navegador que estamos usando HTML5 (versão atual da linguagem). Dentro da tag `<html>`, definimos o atributo `lang` para o idioma (por exemplo, `pt-BR` para português brasileiro) como parte das boas práticas de acessibilidade e SEO. A seção `<head>` contém metadados e configurações (como o `<meta charset="UTF-8">` para definir a codificação de caracteres como UTF-8, garantindo suporte a acentuação e caracteres especiais). A seção `<body>` contém o conteúdo visível da página.

Elementos HTML Essenciais: Abaixo, listamos algumas tags HTML fundamentais e sua finalidade:

Tag (Elemento)	Finalidade / Conteúdo que envolve
`<h1> ... <h6>`	Títulos e subtítulos (h1 é o mais importante, h6 o menos).
`<p>`	Parágrafo de texto.
`<a>`	Hiperlink (âncora). Usado com atributo `href` para links.
`<img>`	Imagem embutida. Requer atributo `src` (fonte da imagem) e **alt** (texto alternativo descritivo).

Tag (Elemento)	Finalidade / Conteúdo que envolve
`<ul>` `<ol>` & `<li>`	Listas não ordenadas (`ul`) ou ordenadas (`ol`) com itens de lista (`li`).
` `	Quebra de linha. *(Elemento vazio, não tem tag de fechamento.)*
`<div>`	Divisão genérica de bloco (usada para agrupar outros elementos).
`<span>`	Elemento genérico em linha (usado para estilizar parte de um texto).

Dicas: - Tags podem possuir **atributos** que fornecem informações adicionais. Ex: `href="https://..."` em um link `<a>`; `src="imagem.png"` e `alt="Descrição da imagem"` em `<img>`. Sempre forneça um atributo **alt** significativo para imagens, pois é obrigatório para um código semântico e acessível [2] . - Nem todas as tags exigem fechamento (tags vazias como `<br>` e `<img>` não possuem tag de fechamento). Na maioria, porém, você abre `<tag>` e fecha com `</tag>` envolvendo o conteúdo. - HTML é case-insensitive (não diferencia maiúsculas de minúsculas nas tags), mas por convenção usamos tudo em **minúsculas** para manter a consistência e legibilidade do código. - Organize o HTML de forma indentada e hierárquica para refletir a estrutura da página e facilitar a leitura.

Exemplo Simples: Estrutura mínima de uma página HTML com um título e um parágrafo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8"> <!-- Define a codificação de caracteres para UTF-8
-->
  <title>Meu Site Exemplo</title>
<!-- Título que aparece na aba do navegador -->
</head>
<body>
  <h1>Olá, mundo!</h1> <!-- Título de nível 1 -->
  <p>Bem-vindo ao meu site de exemplo.</p> <!-- Parágrafo de texto -->
</body>
</html>
```

Comentário: No código acima, temos o **DOCTYPE** declarando HTML5, o elemento raiz `<html>` com idioma, o `<head>` incluindo meta charset e o `<title>`, e o `<body>` com conteúdo. O resultado seria um título grande "Olá, mundo!" e abaixo um texto de boas-vindas.

Exercícios

1. **Página HTML Básica:** Crie um arquivo HTML chamado `index.html` contendo a estrutura básica (DOCTYPE, html, head, body). Dentro do body, inclua um título principal apresentando o nome do site ou da pessoa, um parágrafo curto de apresentação e um link apontando para uma página fictícia de contato (use `href="#"` como placeholder). Certifique-se de incluir uma imagem de logo usando `<img>` com um texto alternativo apropriado.

2. **Sugestão de Solução:** Um possível `index.html`:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Site de Exemplo</title>
</head>
<body>
  <h1>Bem-vindo ao Site de Exemplo</h1>
  <p>Olá! Meu nome é Ana e este é meu site de portfólio.</p>
  <p>Veja mais na <a href="#">página de Contato</a>.</p>
  <img src="logo.png" alt="Logo do Site de Exemplo">
</body>
</html>
```

Explicação: O código define a estrutura básica. Dentro do body, utilizamos `<h1>` para o título principal, `<p>` para textos, `<a>` para link (com href vazio apenas para exemplo) e `<img>` para uma imagem de logo com atributo alt descritivo.

HTML Intermediário

Formulários e Inputs: Formulários HTML permitem coletar dados do usuário. São criados com `<form>` definindo geralmente os atributos `action` (a URL ou página para onde os dados serão enviados) e `method` (método de envio, geralmente `"GET"` ou `"POST"`). Dentro de um form, usamos diversos elementos de controle: - `<input>`: campo de entrada genérico. O atributo `type` define o tipo (texto, senha, email, número, data, etc.). Por exemplo, `<input type="text">` para texto livre, `<input type="email">` para e-mail (validação básica de formato), `<input type="checkbox">` para caixa de seleção, `<input type="radio">` para opção exclusiva, `<input type="file">` para upload, etc. - `<label>`: rótulo descritivo para um campo. Associa-se ao input via atributo `for` (que deve bater com o `id` do input correspondente). Usar `<label>` é importante para acessibilidade e melhora a área clicável dos campos. - Campos específicos: `<textarea>` para texto multilinha; `<select>` com `<option>` para listas suspensas (combobox). - Botões: `<button>` ou `<input type="submit">` para enviar o formulário; `<input type="reset">` para limpar campos (menos usado). - Atributos úteis: `name` (atributo obrigatório que dá nome ao campo, para identificação dos dados no back-end), `placeholder` (texto de dica dentro do campo), `required` (indica que o preenchimento é obrigatório, fazendo o navegador validar antes de enviar).

Tabelas: Apesar de hoje não serem usadas para layout, tabelas são úteis para apresentar dados tabulares (planilhas, resultados). Usamos `<table>` para criar uma tabela, subdividindo em linhas `<tr>` (table row) e células `<td>` (table data). Também podemos definir cabeçalhos de coluna com `<th>` (table header) dentro de um `<tr>`. Exemplos básicos:

```
<table>
  <tr>
    <th>Produto</th><th>Preço</th>
  </tr>
  <tr>
    <td>Caneta</td><td>R$ 2,00</td>
  </tr>
  <tr>
```

```
    <td>Caderno</td><td>R$ 15,00</td>
  </tr>
</table>
```

Esse código geraria uma tabela com duas colunas (Produto e Preço) e duas linhas de dados. Podemos adicionar o atributo `border="1"` no `<table>` para visualizar bordas simples durante testes (embora em produção seja preferível estilizar via CSS).

Elementos de Mídia: HTML permite inserir mídias diversas: - Imagens: já vimos `<img>`. - Áudio: `<audio src="som.mp3" controls>Seu navegador não suporta áudio</audio>` (o texto interno é mostrado caso o navegador não suporte o elemento). - Vídeo: `<video src="video.mp4" controls width="600">Seu navegador não suporta vídeo</video>`. Podemos incluir múltiplas fontes com `<source>` dentro de `<video>` para formatos diferentes, além de legendas com `<track>`. - Iframe: `<iframe src="pagina.html"></iframe>` insere uma página dentro de outra (usado para conteúdo de terceiros como mapas, vídeos do YouTube, etc.). Deve ser usado com cuidado e somente quando necessário.

Outras Práticas Importantes: - **Meta Tags:** No `<head>`, além do charset, podemos incluir `<meta name="description">` com uma descrição breve da página (usado por mecanismos de busca), `<meta name="viewport">` para controle de layout em dispositivos móveis (ex: `content="width=device-width, initial-scale=1.0"` é essencial para *design responsivo* em mobile). - **Classes e IDs:** Qualquer elemento HTML pode ter um atributo `id` (identificador único na página) ou `class` (categoria que pode ser compartilhada por vários elementos). Essas identificações são primordiais para aplicar estilos CSS e manipular via JavaScript. Por exemplo: `<p id="bemvindo" class="destacado">...</p>`. No CSS, `#bemvindo` poderia referir esse elemento único, e `.destacado` poderia estilizar esse e outros elementos semelhantes. - **Melhorias do HTML5:** Novos tipos de input (email, URL, número, etc.) que facilitam validações; atributo `required` e `pattern` (expressão regular para validação customizada) que dão validação no lado do cliente; elementos de mídia mencionados acima; e os elementos semânticos (discutidos adiante) que melhoram a estrutura do documento.

Exemplo de Formulário Simples: Vamos criar um pequeno formulário de contato em HTML:

```
<form action="/enviar_contato.php" method="POST">
  <label for="nome">Nome:</label>
  <input type="text" id="nome" name="nome" placeholder="Seu nome" required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" placeholder="seu@email.com"
required>

  <label for="msg">Mensagem:</label>
  <textarea id="msg" name="mensagem" rows="4" placeholder="Escreva sua
mensagem"></textarea>

  <button type="submit">Enviar</button>
</form>
```

Comentário: Neste exemplo, o formulário usa `method="POST"` (envio não visível na URL) para a página `enviar_contato.php`. Incluímos `label` para cada campo para melhorar a acessibilidade. Os inputs de texto possuem placeholders e estão marcados como `required` para exigir preenchimento. O botão de envio é um `<button type="submit">`. Em um cenário real, no arquivo PHP indicado pelo action, os dados `$_POST['nome']`, `$_POST['email']`, etc., seriam processados.

Exercícios

1. **Tabela de Horários:** Crie uma tabela HTML para organizar seus horários de estudo da semana. Use colunas para os dias (segunda a sexta, por exemplo) e linhas para turnos (manhã, tarde, noite). Preencha algumas células com conteúdos fictícios (ex: "Estudar HTML", "Trabalho", "Lazer"). Inclua um `<thead>` para o cabeçalho da tabela (com os dias) e `<tbody>` para o conteúdo principal, apenas como prática de semântica em tabela.

2. **Sugestão de Solução:**

```
<table border="1">
  <thead>
    <tr>
      <th>Horário</th>
      <th>Segunda</th>
      <th>Terça</th>
      <th>Quarta</th>
      <th>Quinta</th>
      <th>Sexta</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Manhã</td>
      <td>Estudar HTML</td>
      <td>Trabalho</td>
      <td>Trabalho</td>
      <td>Estudar JavaScript</td>
      <td>Trabalho</td>
    </tr>
    <tr>
      <td>Tarde</td>
      <td>Trabalho</td>
      <td>Faculdade</td>
      <td>Faculdade</td>
      <td>Trabalho</td>
      <td>Faculdade</td>
    </tr>
    <tr>
      <td>Noite</td>
      <td>Projeto Pessoal</td>
      <td>Estudar PHP</td>
      <td>Lazer</td>
      <td>Projeto Pessoal</td>
      <td>Lazer</td>
```

```
    </tr>
  </tbody>
</table>
```

Explicação: No exemplo, usamos `<thead>` para separar o cabeçalho (primeira linha com os dias) e `<tbody>` para o conteúdo. Isso não muda visualmente sem CSS, mas deixa o HTML mais estruturado. O atributo `border` foi usado apenas para visualizar a tabela facilmente (em produção, usaria CSS). As células `<td>` foram preenchidas com atividades de exemplo.

3. **Formulário de Cadastro:** Crie um formulário de registro de usuário com campos para **nome**, **email**, **senha** e **confirmação de senha**. Utilize inputs dos tipos adequados (`text` para nome, `email` para email, `password` para senha). Inclua rótulos (`label`) e aplique o atributo `required` em todos os campos. Adicione também um campo do tipo checkbox para aceitar os "Termos de Uso" (com um label "Aceito os termos") e um botão de envio.

4. **Sugestão de Solução:**

```
<form action="/cadastrar.php" method="POST">
  <label for="nome">Nome:</label>
  <input type="text" id="nome" name="nome" required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <label for="senha">Senha:</label>
  <input type="password" id="senha" name="senha" required>

  <label for="senha2">Confirme a Senha:</label>
  <input type="password" id="senha2" name="senha_confirm" required>

  <label>
    <input type="checkbox" name="termos" required>
    Aceito os termos de uso
  </label>

  <button type="submit">Registrar</button>
</form>
```

Explicação: Este formulário de cadastro utiliza campos adequados e exige que todos sejam preenchidos (`required`). A confirmação de senha tem um `name` diferente (`senha_confirm`). O checkbox está dentro de um label para que o texto seja clicável junto com a caixa. Em um cenário real, o script `/cadastrar.php` receberia esses dados via `$_POST` e efetuaria as validações (inclusive verificar se as senhas batem, etc.).

HTML Avançado

HTML Semântico e Estrutura Moderna: Com HTML5, surgiram *elementos semânticos* que melhoram a organização do layout de páginas, substituindo muitos `<div>` genéricos por elementos com significado específico. Usar esses elementos adequadamente torna o código mais **autoexplicativo**, beneficia a **acessibilidade** (por exemplo, leitores de tela entendem a estrutura) e **SEO** (motores de

busca valorizam conteúdo dentro de tags significativas) [3]. Principais elementos semânticos do HTML5:

Elemento semântico	Descrição e Uso
`<header>`	Cabeçalho da página ou de uma seção, normalmente contém logo, título, menu de navegação.
`<nav>`	Seção de navegação com links (menu do site, links de índice). Deve ser usado para agrupar links de navegação importantes.
`<main>`	Conteúdo principal da página. Cada página deve ter no máximo um `<main>`, representando o conteúdo central, excluindo header, nav, footer, etc.
`<section>`	Seção genérica de conteúdo, tipicamente com um título próprio (`<h2>` etc.). Útil para dividir a página em partes tematicamente relacionadas.
`<article>`	Conteúdo independente e auto-contido, como um artigo de blog, uma notícia ou post de fórum. Pode ser lido isoladamente, e geralmente pode conter seu próprio header/footer internos.
`<aside>`	Conteúdo à parte, relacionado ao principal, mas não essencial. Ex: barras laterais com links, infoboxes, anúncios, etc.
`<footer>`	Rodapé da página ou de uma seção. Contém informações de rodapé como créditos, links auxiliares, copyright, etc.

Além desses, existem `<figure>` (agrupa mídia e legenda), `<figcaption>` (legenda de figura), `<mark>` (destacar texto), `<time>` (data/horário), entre outros elementos introduzidos no HTML5 para enriquecer a semântica. Recomenda-se **usar elementos semânticos sempre que possível**, ao invés de inúmeros `<div>` aninhados apenas com classes. Isso torna o HTML mais *significativo* e ajuda tecnologias assistivas a interpretarem corretamente a hierarquia do conteúdo [4] [5].

Boas Práticas Modernas de HTML: - **Semântica sobre Estética:** Evite tags puramente apresentacionais de versões antigas, como `<font>`, `<center>`, `<b>` (use `<strong>` para indicar importância) ou `<i>` (use `<em>` para ênfase). Em vez de usar `<div>` para tudo, utilize os elementos semânticos adequados. Código semântico é mais fácil de manter e estilizar, e garante acessibilidade. Por exemplo, não use um `<div>` clicável como botão; use `<button>` ou `<a>` conforme a semântica (ação x navegação). - **Acessibilidade:** Sempre forneça texto alternativo para imagens (`alt`), legendas para conteúdo audiovisual (via `<track>` se possível), e use etiquetas `<label>` em formulários. Organize títulos hierarquicamente (`<h1>` uma vez por página, `<h2>` para seções principais, etc.) de forma consistente para que leitores de tela e indexadores entendam a estrutura lógica. Um HTML bem estruturado já melhora significativamente a acessibilidade e SEO, e evita "gambiarras" que prejudicam manutenção. - **SEO (Search Engine Optimization):** Além da estrutura de títulos e uso de tags semânticas, inclua meta descrição (`<meta name="description" content="...">`) única por página com um resumo do conteúdo. Use URLs amigáveis (essa parte é mais do lado servidor, mas reflete nos links em `<a>`). **Conteúdo relevante dentro de tags semânticas tende a rankear melhor** [3]. Por exemplo, palavras-chave em um `<h2>` têm mais peso que dentro de um `<div>` comum. - **Performance e Boas Práticas de Código:** Coloque os arquivos CSS no `<head>` (para que o carregamento da página não tenha flash de conteúdo sem estilo) e scripts JavaScript *no final do* `<body>` ou com atributos `defer/async` no `<head>`. Isso assegura que o HTML seja carregado e exibido antes de executar scripts pesados, melhorando o tempo de carregamento percebido [6]. Evite

quantidade excessiva de elementos desnecessários no HTML (manter o DOM enxuto ajuda performance). - **Caracteres Especiais e Entidades:** Use sempre UTF-8 (via `<meta charset="UTF-8">` que já cobrimos) para suporte a caracteres acentuados. Se precisar usar um símbolo especial que pode conflitar com HTML, use entidades HTML (por exemplo, `&copy;` para ©, `<` para <). Hoje em dia, com UTF-8, é comum simplesmente digitar caracteres unicode diretamente, mas é bom conhecer entidades para alguns casos. - **Omissão de Tags Opcionais:** HTML5 permite omitir algumas tags de fechamento (como `</p>` ou `</li>` em certos contextos) e até todo o `<head>` e `<body>` em casos mínimos, mas **não é recomendado omitir** em documentos reais. Escreva todas as tags para clareza e para evitar comportamentos imprevistos. Por exemplo, sempre inclua as tags `<html>`, `<head>`, `<body>` explicitamente [7]. - **PHP embutido:** Quando for misturar PHP no HTML (template), lembre-se que a tag padrão é `<?php ... ?>` [8]. E uma dica avançada: em arquivos puramente PHP, a tag de fechamento `?>` no final é opcional e frequentemente omitida para evitar problemas com espaços em branco após ela [9].

Exemplo de Layout Semântico: Suponha que vamos estruturar o HTML de uma página de blog com cabeçalho, menu, conteúdo principal com um artigo e uma barra lateral, e rodapé. Um possível esqueleto semântico seria:

```
<body>
  <header>
    <h1>Nome do Blog</h1>
    <nav>
      <ul>
        <li><a href="index.html">Início</a></li>
        <li><a href="sobre.html">Sobre</a></li>
        <li><a href="contato.html">Contato</a></li>
      </ul>
    </nav>
  </header>

  <main>
    <article>
      <header>
        <h2>Título do Artigo</h2>
        <p><small>Publicado em <time datetime="2025-06-01">1º de Junho de
2025</time></small></p>
      </header>
      <p>Conteúdo do artigo ...</p>
      <p>Mais conteúdo ...</p>
      <footer>
        <p>Autor: João da Silva</p>
      </footer>
    </article>

    <aside>
      <h3>Artigos Relacionados</h3>
      <ul>
        <li><a href="#">Tema 1</a></li>
        <li><a href="#">Tema 2</a></li>
      </ul>
```

```
    </aside>
  </main>

  <footer>
    <p>&copy; 2025 - Meu Blog. Todos os direitos reservados.</p>
  </footer>
</body>
```

Comentário: Observe o uso de `<header>` no topo da página contendo o título do site e o menu de navegação dentro de `<nav>`. Depois, temos o `<main>` englobando a área principal. Dentro dele, um `<article>` representa uma postagem, que possui seu próprio `<header>` (com título do artigo e data dentro de `<time>`), conteúdo em `<p>` e um `<footer>` com informações do autor. Ao lado, um `<aside>` traz uma seção lateral (no layout, poderia ser coluna lateral) com artigos relacionados. Por fim, o `<footer>` geral da página contém direitos autorais. Essa estrutura, além de semanticamente rica, facilita estilização via CSS (podemos estilizar facilmente cada região) e melhora a experiência de usuários e mecanismos de busca.

Mais Recursos Avançados: HTML oferece outras funcionalidades avançadas como **microdata/JSON-LD** para dados estruturados (SEO avançado), atributos ARIA (`aria-*`) para melhorar acessibilidade em widgets dinâmicos, elementos de template (`<template>` para conteúdo HTML que não renderiza até ser ativado via JS), *Web Components* (com `<slot>` e Shadow DOM), etc. Esses tópicos vão além da revisão geral, mas vale saber que existem. Em suma, dominar HTML avançado é mais sobre **escrever um HTML bem estruturado e semântico**, seguindo as melhores práticas, do que memorizar inúmeras tags. Sempre priorize clareza, semântica e simplicidade na marcação HTML.

Exercícios

1. **Estrutura Semântica de Página:** Desenhe (mentalmente ou no papel) a estrutura de uma página web típica de uma empresa: cabeçalho com logo e menu, seção de banner inicial, seção de serviços, seção "sobre nós", e rodapé com contato. Agora, escreva o HTML semântico dessa página, usando os elementos adequados (`<header>`, `<nav>`, `<section>`, `<article>` se necessário, `<footer>`, etc.). Não é preciso adicionar conteúdo real, concentre-se na hierarquia das tags.
2. **Sugestão de Solução (esboço de estrutura):**

```
<header>
  <h1>Empresa XYZ</h1>
  <nav> ... menu ... </nav>
</header>
<main>
  <section id="banner">
    <h2>Bem-vindo!</h2>
    <p>... texto de boas-vindas ...</p>
  </section>
  <section id="servicos">
    <h2>Serviços</h2>
    <article>
      <h3>Serviço 1</h3>
      <p>Descrição do serviço 1...</p>
    </article>
```

```
    <article>
      <h3>Serviço 2</h3>
      <p>Descrição do serviço 2...</p>
    </article>
  </section>
  <section id="sobre">
    <h2>Sobre Nós</h2>
    <p>Informações sobre a empresa...</p>
  </section>
</main>
<footer>
  <section id="contato">
    <h2>Contato</h2>
    <p>Email: contato@empresa.com</p>
    <p>Telefone: (11) 1234-5678</p>
  </section>
  <p>&copy; 2025 Empresa XYZ</p>
</footer>
```

Explicação: A solução organizada contém um `<header>` com título e nav. Dentro de `<main>`, há várias seções: um banner inicial, serviços (com dois `<article>` representando cada serviço individual, já que poderiam ser independentes), e uma seção "sobre". No `<footer>`, incluímos uma sub-seção de contato (poderia ser um formulário de contato real) e informações de copyright. Note o uso de IDs para seções, que podem ser usados para âncoras de navegação ou estilos. Essa estrutura deixa claro o papel de cada parte da página.

3. **Revisando Semântica:** Abaixo está um trecho de HTML escrito de forma não semântica e pouco acessível. Reescreva-o corrigindo as tags para torná-lo semântico e acessível, sem alterar o conteúdo visual.

```
<div style="text-align:center; font-size: 20px;">Notícias do Dia</div>
<p><b>01/01/2025</b> - Empresa XYZ lança novo produto<br>
<i>Os detalhes do lançamento incluem ...</i></p>
```

4. **Sugestão de Solução:**

```
<h2 style="text-align:center;">Notícias do Dia</h2>
<article>
  <header>
    <h3>Empresa XYZ lança novo produto</h3>
    <p><small><time datetime="2025-01-01">01/01/2025</time></small></p>
  </header>
  <p>Os detalhes do lançamento incluem ...</p>
</article>
```

Explicação: Transformamos o título "Notícias do Dia" em um `<h2>` apropriado (e removemos o uso de `<div>` puramente visual). A notícia em si vira um `<article>` com um `<header>` contendo título (`<h3>` para essa notícia específica) e a data formatada adequadamente dentro de `<time>` (com atributo `datetime` para padrões globais). O texto descritivo permanece em `<p>` normal (removendo o `<i>` desnecessário; se quiséssemos enfatizar alguma parte do

texto, usaríamos `<em>`). Também eliminamos `<br>` para quebra de linha desnecessária, estruturando com parágrafos adequados. O CSS em linha para centralizar texto e tamanho de fonte idealmente iria para um arquivo CSS externo, mas mantivemos o `text-align:center` no `<h2>` apenas para preservar a aparência de forma simples aqui, enfatizando que a estilização deve ser separada da estrutura.

CSS

CSS Iniciante

Conceito Geral: CSS (Cascading Style Sheets, ou *Folhas de Estilo em Cascata*) é a linguagem utilizada para estilizar o HTML, controlando aparência e layout. Enquanto o HTML fornece a estrutura e o conteúdo, o CSS cuida de cores, fontes, espaçamentos, posicionamento e decoração visual do site. O termo "cascading" (cascata) refere-se à forma como as regras são aplicadas, permitindo que estilos mais específicos ou definidos posteriormente sobrescrevam os anteriores, seguindo certas prioridades.

Como adicionar CSS a uma página: Há três formas principais: - **Arquivo Externo:** É a prática recomendada. Você cria um arquivo `.css` separado (por exemplo, `styles.css`) e o inclui no HTML com a tag `<link>` dentro do `<head>`, assim: `<link rel="stylesheet" href="styles.css">`. Isso mantém o conteúdo separado da apresentação e facilita a manutenção e reuso de estilos em múltiplas páginas. - **CSS Interno (Embutido no HTML):** Utiliza-se uma tag `<style>` dentro do `<head>` do documento HTML para escrever regras CSS diretamente. Exemplo:

```
<style>
  /* CSS interno */
  body { background: #fff; }
  h1 { color: blue; }
</style>
```

Funciona em casos simples ou testes rápidos, mas para projetos maiores é melhor usar arquivo externo. - **Inline (no atributo style):** Aplica estilos diretamente em elementos via atributo `style`, e.g., `<p style="color: red; text-align: center;">Texto</p>`. **Não é recomendado** em geral, pois mistura HTML com design e dificulta a manutenção (deve-se evitar estilos inline a não ser que seja absolutamente necessário para sobrepor algo específico). Prefira classes e um CSS centralizado.

Sintaxe CSS: Uma regra CSS básica consiste em um **seletor** seguido de um bloco de declarações entre chaves `{ }`. Dentro, escrevemos pares **propriedade: valor** terminados por ponto-e-vírgula `;`. Exemplo:

```
h1 {
  color: blue;
  font-size: 30px;
}
```

Aqui `h1` é o seletor (alvo da regra, nesse caso todos os `<h1>` da página), `color` é uma propriedade CSS e `blue` seu valor (tornando o texto azul). `font-size: 30px;` define o tamanho da fonte. Podemos listar várias propriedades para o mesmo seletor.

Seletores Básicos: - **Por Tag (Tipo):** Seleciona elementos pelo nome da tag. Ex: `p { ... }` estiliza todos `<p>`. - **Por Classe:** Seleciona elementos que possuem determinada classe. Usa-se um ponto antes do nome da classe. Ex: `.destacado { background: yellow; }` afeta elementos que em HTML tenham `class="destacado"`. - **Por ID:** Seleciona um elemento específico com certo id. Usa-se `#`. Ex: `#menu { font-weight: bold; }` estiliza o elemento que tiver `id="menu"`. *Dica:* Id deve ser único por página, então #menu refere-se a no máximo um elemento. - **Seletores combinados:** Podemos combinar seletores para precisar o alvo. Por exemplo, `ul.lista-produtos li { ... }` estiliza apenas `<li>` que esteja dentro de um `<ul class="lista-produtos">`. Ou `.card p { ... }` estiliza `<p>` dentro de qualquer elemento com class "card". - **Seletores universais e por atributo:** O seletor universal `*` seleciona todos os elementos. Seletores de atributo, como `input[type="text"] { ... }`, estilizam elementos com um determinado atributo/valor.

Cascata e Especificidade (conceito breve): Se duas regras conflitarem (por exemplo, ambas alteram a mesma propriedade de um elemento), a *cascata* define qual vale. Regras mais específicas (ID > classe > tag) tendem a prevalecer sobre genéricas. Além disso, a última definição no CSS sobrescreve definições anteriores de mesma especificidade. Por exemplo, se você tem:

```
p { color: black; }
#intro p { color: red; }
```

Um `<p>` dentro de um elemento com id "intro" ficará vermelho devido à regra mais específica `#intro p`. A *cascata* e especificidade são assuntos amplos, mas lembre-se: **escreva CSS de forma que regras específicas sejam necessárias somente onde preciso**, e evite usar `!important` (recurso que força prioridade mais alta) a menos que inevitável.

Propriedades CSS Comuns (iniciais): - *Texto e Fontes:* `color` (cor do texto), `font-size` (tamanho da fonte, e.g., `16px`), `font-family` (tipo de fonte, e.g., `font-family: Arial, sans-serif;`), `font-weight` (peso/negrito), `text-align` (alinhamento do texto), `text-decoration` (decoração como underline, etc., frequentemente usada para remover sublinhado de links com `text-decoration: none`). - *Cores:* Podem ser especificadas por nome (e.g., `red`), por código hexadecimal (e.g., `#FF0000` para vermelho), por função RGB/RGBA (`rgb(255,0,0)` ou com alfa `rgba(255,0,0,0.5)`), ou HSL. Cores em CSS são flexíveis, apenas mantenha um formato consistente e use contraste adequado para acessibilidade. - *Planos de fundo:* `background-color` (cor de fundo), `background-image` (imagem de fundo), `background-repeat`, etc. Ex: `body { background-color: #f0f0f0; }`. - *Box model (introdução):* Elementos HTML podem ser visualizados como caixas retangulares. O **modelo de caixa** consiste em **margem (margin)**, **borda (border)**, **padding (recheio interno)** e o **conteúdo** em si. Por ora, saiba que `margin` controla espaçamento externo do elemento, e `padding` o espaçamento interno entre seu conteúdo e sua borda. Por exemplo, `padding: 10px; margin: 5px; border: 1px solid #000;` em um `<div>` dará preenchimento interno de 10px, margem externa de 5px e borda de 1px sólida preta.

Exemplo simples de CSS aplicado: Vamos estilizar a página HTML do exemplo anterior.

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

Em um arquivo **styles.css** externo, poderíamos ter:

```css
/* styles.css */
body {
  background-color: #EFEFEF; /* cor de fundo suave */
  font-family: Arial, sans-serif; /* fonte padrão */
}
h1 {
  color: #2E8B57; /* verde */
  text-align: center; /* centraliza o título */
}
p {
  font-size: 16px;
  line-height: 1.5; /* espaçamento entre linhas para melhorar legibilidade */
}
/* Estilizando o link */
a {
  color: #0066CC;
  text-decoration: none; /* remove sublinhado padrão */
}
a:hover {
  text-decoration: underline; /* sublinha o link ao passar o mouse */
}
```

Comentário: O CSS acima define um fundo cinza claro para a página, aplica uma fonte padrão genérica, centraliza e colore o `<h1>` de verde (código hexadecimal #2E8B57), define tamanho e espaçamento de linha para textos de parágrafo, e estiliza links para ficarem azul escuro sem sublinhado, sublinhando-os apenas no hover (quando o usuário passa o cursor). Repare no uso de pseudo-classe `:hover` para estilizar o estado do link interativo. Separar estilos em classes permite reuso: poderíamos, por exemplo, criar `.destaque { background: yellow; }` e adicionar `class="destaque"` em qualquer elemento HTML para destacar com fundo amarelo.

Exercícios

1. **Aplicando Estilos Básicos:** Considere o HTML criado no exercício 1 da seção HTML Iniciante (a página básica com título, parágrafo, link e imagem). Crie um arquivo CSS para estilizar essa página com as seguintes especificações: fundo da página em cor suave, título centralizado e de cor diferente, parágrafo justificado e com espaçamento entre linhas de 1.6, o link em uma cor que combine e que ao passar o mouse fique sublinhado. Inclua também um estilo para a imagem, por exemplo, limitando sua largura a 100% do contêiner (para torná-la responsiva). Vincule o CSS externamente no HTML.
2. **Sugestão de Solução:**

```css
/* estilo.css */
body {
  background-color: #FAFAFA;
  color: #333;
  font-family: sans-serif;
  margin: 20px;
}
h1 {
```

```
  text-align: center;
  color: #4CAF50; /* verde principal */
}
p {
  text-align: justify;
  line-height: 1.6;
}
a {
  color: #0088CC;
  text-decoration: none;
}
a:hover {
  text-decoration: underline;
}
img {
  max-width: 100%;
  height: auto;
  display: block;
  margin: 0 auto; /* centraliza a imagem horizontalmente */
}
```

Explicação: Definimos um fundo bem claro no `body` e cor de texto padrão #333 (um cinza escuro, menos cansativo que preto puro). O margin de 20px no body dá espaçamento das bordas da janela. O `h1` centralizado em verde (usamos um tom verde do material design, #4CAF50). Parágrafos justificados com `line-height` para melhorar leitura. Links azuis e sublinhando no hover. A imagem foi tornada **responsiva** com `max-width: 100%` (nunca exceder a largura do contêiner) e `height: auto` para manter proporção, além de `display: block` e margin auto para centralizá-la. Esse CSS deveria ser salvo, por exemplo, em "estilo.css" e incluído no HTML com `<link rel="stylesheet" href="estilo.css">` dentro do `<head>`.

3. **Classe de Destaque:** No seu site, suponha que você queira destacar algumas palavras importantes em vermelho e negrito sem usar as tags `<b>` ou `<span style="...">` inline em cada caso. Crie uma classe CSS chamada `.destaque` que deixe o texto em negrito (`font-weight: bold`) e cor vermelha. Depois, demonstre seu uso aplicando essa classe em um pequeno trecho de HTML de exemplo (por exemplo, um parágrafo com algumas palavras marcadas com `<span class="destaque">...</span>`).
4. **Sugestão de Solução:**

```
.destaque {
  color: red;
  font-weight: bold;
}
```

Exemplo de uso no HTML:

```
<p>Esta frase contém uma <span class="destaque">palavra muito
importante</span> que está destacada.</p>
```

Explicação: A classe `.destaque` definida no CSS pinta o texto de vermelho e o deixa em negrito. No HTML, aplicamos essa classe a um `<span>` envolvendo apenas a palavra (ou expressão) que deve ganhar destaque, mantendo assim a semântica do restante do texto intacta (não estamos usando `<b>` de forma indiscriminada, e `<span>` por si só não tem significado semântico, mas aqui serve como contêiner para estilização). Esse método é muito utilizado para destacar trechos sem alterar a estrutura HTML geral.

CSS Intermediário

Modelo de Caixa (Box Model) em Detalhe: Todos os elementos em bloco no HTML (como `<div>`, `<p>`, `<section>`, etc.) podem ser imaginados como caixas retangulares. O box model define: - **Conteúdo (content):** A área contendo o conteúdo (texto, imagem). - **Padding:** Espaço interno entre o conteúdo e a borda (cria "acolchoamento" dentro do elemento, com a cor de fundo do elemento visível abaixo do padding). - **Border:** Borda do elemento, com espessura, estilo (sólido, tracejado, etc.) e cor configuráveis (ex: `border: 2px solid black;`). - **Margin:** Espaço externo, do lado de fora da borda, que separa o elemento dos vizinhos.

A propriedade `box-sizing` controla se `width` / `height` incluem ou não o padding e border no cálculo. Por padrão (`box-sizing: content-box`), a largura definida aplica-se apenas ao conteúdo, e padding/border aumentam o tamanho total. Se usar `box-sizing: border-box`, então a largura total do elemento inclui padding e border, facilitando cálculos de layout. Muitas vezes os desenvolvedores aplicam `* { box-sizing: border-box; }` globalmente para tornar o comportamento mais intuitivo.

Display e Fluxo dos Elementos: Por padrão, elementos HTML têm um comportamento de display: - **block (bloco):** Ocupa toda a largura disponível, começando em nova linha. Ex: `<div>, <p>, <section>`. Podemos definir largura/altura manualmente. Margens verticais se somam normalmente. - **inline (em linha):** Flui junto com o texto, só ocupa o espaço necessário ao seu conteúdo. Ex: `<span>, <a>, <strong>`. Não aceita definir width/height ou margin-top/bottom (essas propriedades não afetam elementos inline). - **inline-block:** Combina características: flui em linha como texto, porém se comporta como bloco internamente (aceita width, height, margin). Ex: `<img>` se comporta como inline-block por padrão (pode definir width/height e ele aparece ao lado de texto). - **none:** Elemento não é renderizado (invisível e sem ocupar espaço). Usado para ocultar elementos (`display: none`). - *Outros:* `list-item` (comportamento de item de lista, ex `<li>` padrão), `table`, etc., mas raramente precisaremos setar manualmente esses além de resets ou situações específicas.

Comparação display block vs inline:

Tipo display	Comportamento	Exemplo de elementos
block	Ocupa linha toda; largura 100% por padrão; começa em nova linha; pode ter largura, altura, margin/padding em todos os lados.	`div, p, section, article, header, footer` (por padrão)

Tipo display	Comportamento	Exemplo de elementos
inline	Ocupa apenas o espaço do conteúdo; não quebra linha automaticamente (pode ter vários inline lado a lado); *ignora* propriedades de tamanho explícito; margens verticais não aplicadas (horizontais sim, mas cuidadocom espaçamento de texto).	`span, a, strong, em, img*` (`img` é inline replacer)
inline-block	Como inline, mas respeita width/height/ margins. Não quebra linha mas podemos controlar seu tamanho. Útil para criar elementos em linha customizados com tamanho fixo.	`img` (efetivamente inline-block), `button` (alguns navegadores tratam como inline-block) ou elementos que definirmos com `display:inline-block`.

Posicionamento de Elementos: CSS oferece a propriedade `position` para tirar elementos do fluxo normal ou posicioná-los de forma mais controlada: - **static:** Padrão de todos, segue o fluxo normal da página. (Nem precisa declarar, é o default.) - **relative:** O elemento permanece no fluxo, **mas podemos deslocá-lo** levemente usando `top`, `left`, `right`, `bottom` relativo à sua posição original. Útil para pequenos ajustes ou para criar um contexto para elementos posicionados internamente (um elemento **absoluto** dentro de um relative é posicionado relativo a esse pai). - **absolute:** Remove o elemento do fluxo normal (outros elementos se comportam como se ele não estivesse lá). Ele é posicionado em relação ao antecessor mais próximo que seja posicionado (i.e., tenha position relative/absolute/fixed) ou senão em relação à página/viewport. Deve-se especificar coordenadas com `top`, `left`, etc. Ex: para criar um pop-up dentro de um container, podemos ter container com `position: relative` e o pop-up filho `position: absolute; top: 10px; left: 10px;`. - **fixed:** Semelhante ao absolute, mas sempre relativo à viewport (janela) e permanece fixo mesmo ao dar scroll. Ex: menu fixo no topo da página: `position: fixed; top: 0; left: 0; width: 100%;`. - **sticky:** Comportamento híbrido (novo no CSS). Um elemento sticky se comporta como relativo até atingir um certo deslocamento na rolagem, daí se torna fixo. Ex: `position: sticky; top: 0;` em um cabeçalho de tabela fará o cabeçalho ficar preso no topo quando você rolar a tabela.

Layout com Flexbox (Flexível): *Flexbox* é um modelo de layout moderno (CSS3) para criar facilmente layouts unidimensionais (em linha ou em coluna) e distribuir espaço. Tornou obsoletas muitas técnicas antigas como usar tabelas ou floats para layout. Para usar: - Defina o contêiner como `display: flex;` (pode também usar `display: inline-flex;` para um flex contido em texto). - Por padrão, os itens flex (filhos diretos) serão organizados em **linha horizontal**. Use `flex-direction: column;` se quiser vertical. - Propriedades do contêiner: - `justify-content`: alinha os itens ao longo do eixo principal (horizontal se flex-direction row). Valores comuns: `flex-start` (início), `flex-end` (fim), `center`, `space-between` (espaço entre itens, extremidades encostadas), `space-around` (espaços iguais ao redor). - `align-items`: alinha itens no eixo transversal (vertical se linha). Ex: `stretch` (padrão, itens esticam para preencher altura do contêiner se não tiver altura própria), `center` (centraliza verticalmente), `flex-start` / `flex-end` (topo ou base). - `gap`: espaçamento consistente entre os itens (ex: `gap: 20px;` coloca 20px entre cada item, horizontalmente e/ou verticalmente conforme o eixo). - Propriedades dos itens flex (aplicadas em cada filho): - `flex-grow` e `flex-shrink`: controlam se e como o item cresce ou encolhe para preencher espaço sobrando ou caber em espaço reduzido. - `flex-basis`: tamanho inicial pretendido do item antes de distribuir espaço (pode ser em px, %, etc., ou `auto`). - *Atalho:* `flex` é abreviação que engloba grow, shrink e basis juntos (ex: `flex: 1 1 200px;`). - `align-self`: permite sobrepor o `align-items` do contêiner para um item específico (ex: um item pode ser alinhado diferente dos outros).

Um exemplo simples de **layout com flex**: vamos criar um container com três caixas lado a lado de igual largura:

```
<div class="container">
  <div class="caixa">Caixa 1</div>
  <div class="caixa">Caixa 2</div>
  <div class="caixa">Caixa 3</div>
</div>
```

CSS:

```
.container {
  display: flex;
  justify-content: space-between; /* espaçamento entre as caixas */
}
.caixa {
  background: #EEE;
  padding: 20px;
  flex: 1; /* cada caixa cresce igualmente para ocupar espaço disponível */
  margin: 5px;
}
```

Nesse exemplo, `.container` é flex, alinhando itens com espaçamento igual entre eles. Cada `.caixa` tem `flex: 1`, equivalendo a `flex: 1 1 0` por padrão, ou seja, todas crescem igualmente dividindo o espaço. As margens de 5px dão um respiro externo e o padding de 20px dá espaçamento interno.

Float e Clear (legado): Antes do flexbox e grid, usava-se muito `float` para criar colunas. Um elemento com `float: left` sai do fluxo normal e "flutua" à esquerda, permitindo que conteúdo seguinte (como texto) flua ao seu lado. Flutuadores precisam em algum ponto ser "limpos" (clear) para que o container pai reconheça sua altura. Isso era feito adicionando `clear: both` em um elemento logo depois, ou via CSS hack (clearfix). Atualmente, *floats* ainda são úteis para posicionar imagens dentro de texto (ex: uma imagem float left faz o texto rodeá-la à direita), mas para layout inteiro de página são desnecessários graças ao flexbox e grid. Conheça float/clear para manutenção de código legado, mas prefira as técnicas modernas para novos projetos.

Media Queries (Responsividade): Para tornar layouts responsivos (adaptarem-se a diferentes tamanhos de tela), usamos media queries no CSS. A sintaxe básica:

```
@media (max-width: 600px) {
  /* CSS aqui dentro vale apenas para visores com largura até 600px */
  body {
    background: yellow;
  }
}
```

No exemplo, se a tela for menor ou igual a 600px de largura (um smartphone em modo retrato geralmente), o fundo ficará amarelo; em telas maiores, essa regra não se aplica. Podemos usar `min-`

`width`, combinar condições (`and`, `or`), direcionar por tipo de dispositivo (screen, print), etc. Uma estratégia comum é *mobile-first*: escrever estilos padrão para telas pequenas, e usar media queries com `min-width` para adicionar/alterar estilos em telas maiores.

Exemplo de Media Query: Suponha que em um site com duas colunas (conteúdo e sidebar), queremos que em telas pequenas a sidebar fique abaixo do conteúdo (layout de uma coluna). Se as duas colunas são elementos .main (conteúdo) e .sidebar, poderíamos ter:

```
/* Estilos padrão (mobile-first): colunas empilhadas */
.main, .sidebar {
  display: block;
  width: 100%;
}
/* Em telas maiores que 768px, lado a lado */
@media (min-width: 768px) {
  .container {
    display: flex;
  }
  .main {
    width: 70%;
  }
  .sidebar {
    width: 30%;
  }
}
```

Aqui, em tela pequena, `.main` e `.sidebar` ocupam 100% (block, empilhados). A partir de 768px, aplicamos display flex no container para colocá-los lado a lado, e definimos porcentagens de largura para cada (70/30). Em flexbox poderíamos também fazer sem precisar essas porcentagens fixas usando `flex` para proporção.

Exercícios

1. **Layout Flex Responsivo:** Crie um layout de galeria de fotos com 4 imagens por linha em telas grandes, que reduza para 2 por linha em telas médias e 1 por linha em telas pequenas. Use um contêiner flex ou grid para implementar isso:
2. Defina um contêiner `.galeria` que seja flexível ou grid.
3. Cada item (por exemplo, `.foto`) deve ter uma imagem ou placeholder e um pequeno texto legenda embaixo.
4. Use media queries para ajustar o número de colunas conforme a largura do dispositivo (pode usar flex-wrap ou mudar a configuração do grid).
5. **Sugestão de Solução (utilizando CSS Grid para praticar):**

```
.galeria {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 10px;
}
.foto {
```

```
  background: #ccc;
  padding: 10px;
  text-align: center;
}
.foto img {
  max-width: 100%;
  height: auto;
  display: block;
  margin: 0 auto 5px auto;
}
@media (max-width: 992px) {  /* telas médias, <= 992px */
  .galeria {
    grid-template-columns: repeat(2, 1fr);
  }
}
@media (max-width: 600px) {  /* telas pequenas, <= 600px */
  .galeria {
    grid-template-columns: 1fr;
  }
}
```

Explicação: Usamos **CSS Grid** para facilidade: `grid-template-columns: repeat(4, 1fr)` cria 4 colunas iguais. O gap de 10px espaça os itens. Em telas até 992px (aprox. tablets), mudamos para 2 colunas; até 600px (celulares), para 1 coluna. Cada `.foto` tem padding e fundo cinza claro só para visualização, e centralizamos texto. As imagens dentro `.foto` são responsivas (`max-width:100%`). Essa solução atinge o pedido de 4, 2, 1 colunas de acordo com o tamanho da tela. Em flexbox, uma alternativa seria usar `flex-wrap: wrap` e ajustar a base/flex dos itens via media queries, mas grid torna mais direto nesse caso.

6. **Centralizando com Flexbox:** Utilize flexbox para centralizar completamente (horizontal e verticalmente) um elemento de destaque na página. Por exemplo, considere um `<div class="popup">Conteúdo</div>` que deve aparecer centralizado no centro da viewport, acima de tudo. Como conseguir isso com CSS?
7. **Sugestão de Solução:**

```
html, body {
  height: 100%; /* garantir que o body tenha altura total */
  margin: 0;
}
body {
  display: flex;
  justify-content: center;
  align-items: center;
}
.popup {
  padding: 20px;
  background: white;
  border: 2px solid #000;
}
```

Explicação: Tornamos o `<body>` um contêiner flex e, crucialmente, damos a ele a altura total da tela (`height: 100%` no html/body). Com `justify-content: center` e `align-items: center`, qualquer conteúdo dentro do body será centralizado nos dois eixos. Assim, `.popup` aparece exatamente no meio da página. Note que removemos margens do body (que por padrão tem alguma margem) para evitar deslocamento. Essa técnica é muito útil para centralizar elementos de forma simples com flex, incluindo cenários como mostrar uma mensagem "Loading..." centralizada, modais, etc.

8. **Hambúrguer Menu Responsivo:** Pense num menu de navegação simples com links horizontais para desktop que deve se tornar um botão "menu" (icone de hambúrguer) em tela pequena. Qual seria a abordagem geral usando CSS (e possivelmente um pouco de JS) para esconder os links e mostrar um botão no mobile, e vice-versa no desktop?
9. **Sugestão de Solução (explicação):** Você poderia usar CSS para controlar a visibilidade: por exemplo, ter um menu:

```
<nav>
  <ul class="menu">
    <li><a href="#">Home</a></li><li>...</li><!-- etc -->
  </ul>
  <button class="menu-toggle">≡ Menu</button>
</nav>
```

No CSS, por padrão em desktop (min-width grande) exibir a lista e esconder o botão, e em mobile (max-width pequeno) fazer o inverso:

```
.menu-toggle { display: none; }
@media (max-width: 600px) {
  .menu { display: none; }
  .menu-toggle { display: block; }
}
```

Assim, em telas pequenas a lista de links some e aparece o botão. O ícone "≡" (três barras) é inserido como conteúdo do botão. Para realmente mostrar/esconder os links ao clicar no botão, seria necessário JavaScript para adicionar uma classe que torne `.menu { display: block; }` quando ativo. Contudo, apenas com CSS é possível conseguir algo com `:target` ou checkbox hack, mas isso foge do escopo *inicial*. O importante é entender que o CSS controla a visibilidade conforme a largura, tornando o menu adaptativo. Essa técnica demonstra o uso combinado de **media queries** e manipulação de `display` para construir interfaces responsivas.

CSS Avançado

CSS Grid Layout: Complementando o flexbox, o **CSS Grid** é um sistema de layout bidimensional, excelente para criar grades e disposições mais complexas em ambos eixos (linhas e colunas). Já utilizamos um exemplo de grid acima. Alguns conceitos chave: - Defina um contêiner com `display: grid;`. - Use `grid-template-columns` e `grid-template-rows` para definir o tamanho/quantidade de colunas e linhas. Pode usar valores fixos (px, %, etc.) ou relativos (`fr` que representa frações do espaço disponível). - `grid-template-columns: repeat(3, 1fr);` = 3 colunas iguais dividindo o espaço. - `grid-template-columns: 200px 1fr 1fr;` = primeira coluna 200px, as outras duas dividem igualmente o restante. - Espaçamento entre células com `gap` (pode ser separado

em `row-gap` e `column-gap`). - Colocar itens em locais específicos: usando propriedades no item como `grid-column: 1 / span 2;` (faz o item ocupar da coluna 1 até ocupar 2 colunas) ou no container definindo áreas nomeadas. - Grid é muito poderoso e flexível; para a revisão, lembre-se de usar grid para layouts em que tanto o alinhamento horizontal quanto vertical importam (por exemplo, galeria de cartão com arranjo regular, layouts de página completos, etc.), enquanto flex é mais simples para alinhar itens numa direção e fazer distribuição proporcional.

Animações e Transições: CSS Avançado inclui adicionar dinamismo visual sem JavaScript: - **Transições (transition):** Permitem animar mudanças graduais de valores CSS. Por exemplo, ao mudar a cor de um botão no hover, podemos suavizar:

```
button {
  background: blue;
  transition: background 0.3s ease;
}
button:hover {
  background: lightblue;
}
```

Isso fará o fundo do botão animar de azul para azul-claro em 0.3 segundos, ao invés de trocar instantaneamente. Podemos especificar múltiplas propriedades e timings, e funções de easing (linear, ease, ease-in, etc.). - **Animações (keyframes):** Com `@keyframes` podemos definir uma sequência de frames CSS e aplicar a um elemento via `animation`. Exemplo simples:

```
@keyframes destaquePisca {
  0%   { background-color: yellow; }
  50%  { background-color: orange; }
  100% { background-color: yellow; }
}
.aviso {
  animation: destaquePisca 1s infinite;
}
```

Aqui definimos uma animação nomeada "destaquePisca" que alterna entre amarelo e laranja. Aplicando `animation: destaquePisca 1s infinite;` em `.aviso`, qualquer elemento com class "aviso" vai ficar pulsando entre as cores. Animações têm muitas propriedades (delay, iteration-count, direction, etc.), mas a ideia é que CSS consegue criar animações desde simples transições de hover até banners animados complexos.

Pré-processadores (Sass, Less) e Organizações de CSS: Em projetos grandes, arquivos CSS podem ficar muito extensos e difíceis de manter. Ferramentas como Sass (Syntactically Awesome Style Sheets) ou Less permitem escrever CSS usando variáveis, aninhamento de seletores, funções e depois compilar para CSS puro. Embora não façam parte do CSS em si, são amplamente usadas na indústria. Exemplo de Sass:

```
$cor-primaria: #3498db;
nav {
  background: $cor-primaria;
```

```
  ul {
    list-style: none;
    li { display: inline-block; margin-right: 20px; }
  }
}
```

Isso quando compilado vira CSS normal, mas a sintaxe facilita repetir valores e estrutura. Outra técnica de organização de CSS é a metodologia **BEM (Block Element Modifier)** para nomenclatura de classes, visando evitar conflitos de escopo. Em BEM, classes são nomeadas como `.bloco__elemento--modificador`. Ex: `.card__titulo--destaque` indica o elemento "título" do bloco "card" em sua variação "destaque". Não é necessário usar BEM, mas é útil em times para padronizar.

Variáveis CSS (Custom Properties): Diferente dos pré-processadores, CSS atualmente suporta variáveis nativas. Definidas com `--nome` e usadas com `var(--nome)`. Geralmente definidas no `:root` para ficarem disponíveis globalmente:

```
:root {
  --cor-fundo: #202020;
  --cor-texto: #FFFFFF;
}
body {
  background: var(--cor-fundo);
  color: var(--cor-texto);
}
```

Vantagem: podem ser atualizadas dinamicamente via JS ou com media queries (por exemplo, mudar um conjunto de cores para modo escuro/claro), e herdam escopo (podemos definir variáveis a nível de um container para temas locais).

Funções CSS úteis: CSS possui funções nativas como `calc()` (cálculos com unidades, ex: `width: calc(100% - 50px);`), `min()` / `max()` / `clamp()` para valores responsivos (clamp permite definir um valor ideal com mínimos e máximos), funções de cor como `rgba()`, `hsl()`, etc., e seletores avançados como `:nth-child()` e `:nth-of-type()` para selecionar elementos de forma específica (ex: `.lista li:nth-child(odd) { background: #eee; }` estiliza itens ímpares de lista com fundo).

Pseudo-elementos: `::before` e `::after` permitem inserir conteúdo virtual antes ou depois de um elemento (usados para efeitos decorativos, ícones, limpadores de float, etc.). Exemplo:

```
button::after {
  content: " →";
}
```

Adicionaria uma setinha após o texto de cada botão via CSS, sem precisar colocar no HTML. Pseudo-elementos combinados com animações e transformações podem criar efeitos avançados.

Dica de Compatibilidade: Sempre teste em diferentes navegadores. Use prefixos CSS (`-webkit-`, `-moz-`, etc.) quando necessário para suportar recursos novos em navegadores mais antigos. Hoje, a maioria dos recursos citados (flex, grid, etc.) tem bom suporte nas versões recentes dos principais navegadores.

Exercícios

1. **Transição Suave:** Adicione uma transição suave de 0.4s em um link que muda de cor ao passar o mouse. Por exemplo, links que são originalmente cinza e ficam pretos no hover. Implemente via CSS e teste em um navegador para ver o efeito.
2. **Sugestão de Solução:**

```
a {
  color: gray;
  transition: color 0.4s ease;
}
a:hover {
  color: black;
}
```

   Explicação: A propriedade `transition: color 0.4s ease;` aplicada ao link significa que quando qualquer mudança na propriedade `color` ocorrer (nesse caso, no :hover mudamos para preto), o navegador fará essa alteração gradualmente ao longo de 0.4 segundos com aceleração suave (`ease`). Assim, em vez do link ficar preto instantaneamente, ele vai escurecendo suavemente quando o ponteiro entra, e voltando a cinza suavemente quando sai.
3. **Keyframe Animation:** Crie uma animação com `@keyframes` que faça um elemento piscar (alternar opacidade de 0 a 1) indefinidamente. Aplique essa animação a um elemento de aviso para chamar atenção.
4. **Sugestão de Solução:**

```
@keyframes blink {
  0%, 100% { opacity: 1; }
  50% { opacity: 0; }
}
.pisca {
  animation: blink 1s infinite;
}
```

   HTML de exemplo: `<p class="pisca">ALERTA: ação necessária!</p>`. *Explicação:* Os keyframes "blink" definem que no início e fim do ciclo (0% e 100%) a opacidade está em 1 (visível), e no meio (50%) está 0 (invisível). Aplicando em `.pisca` a propriedade `animation: blink 1s infinite;`, o elemento ficará piscando continuamente com um ciclo de 1 segundo. Poderíamos adicionar `alternate` para ele piscar e voltar alternando suavemente, mas como usamos valores discretos de opacidade, está ok. Essa técnica é útil para destacar algo urgente visualmente.
5. **Tema com Variáveis CSS:** Suponha que você queira implementar um "modo escuro" no seu site usando variáveis CSS. Descreva como você faria para definir cores em variáveis e depois alternar para modo escuro alterando essas variáveis (por exemplo, adicionando uma classe `dark-mode` no `<html>` ou `<body>`).

6. **Sugestão de Solução:** Primeiro, definir as variáveis de cor padrão (modo claro) no `:root`:

```
:root {
  --bg-color: #FFFFFF;
  --text-color: #000000;
  --primary-color: #4CAF50;
}
body {
  background: var(--bg-color);
  color: var(--text-color);
}
header, footer {
  background: var(--primary-color);
}
```

Agora, para o modo escuro, podemos criar uma variação dessas variáveis sob uma classe que quando presente altera os valores:

```
.dark-mode {
  --bg-color: #121212;
  --text-color: #E0E0E0;
  --primary-color: #90CAF9;
}
```

No HTML, ao adicionar `<body class="dark-mode">` (por exemplo, via um botão toggle com JavaScript), automaticamente as variáveis assumem os novos valores definidos dentro de `.dark-mode` (porque CSS Custom Properties participam da cascata e podem ser redefinidas em escopos). Assim, o background do body ficará #121212, texto claro, e a cor primária mudada (no exemplo, usei um tom azul claro para destacar header/footer no modo escuro). Removendo a classe `dark-mode` volta ao normal. Essa abordagem é elegante, pois você só muda variáveis, sem duplicar todos os seletores e regras de cor para dois temas – facilita manter consistência e realizar o toggle de temas com transições suaves (pode até adicionar `transition` nas propriedades para animar a troca de tema).

JavaScript

JavaScript Iniciante

Visão Geral: JavaScript é a linguagem de programação padrão da web para adicionar interatividade e comportamento dinâmico às páginas. Diferente de HTML e CSS, que são declarativos, JavaScript é uma linguagem de script imperativa que roda, na maior parte das vezes, dentro do navegador web do usuário (lado do cliente). JS pode **manipular o HTML e CSS** via DOM, reagir a eventos do usuário (cliques, teclas), fazer cálculos, validar formulários e até realizar requisições de rede assíncronas (AJAX) sem recarregar a página. Hoje em dia, JavaScript não se limita ao navegador: com runtimes como Node.js, também é usado no servidor, mas aqui focaremos no uso em páginas web.

Inserindo JavaScript em páginas HTML: Existem duas maneiras comuns: - **Externo:** Criar um arquivo `.js` e incluí-lo com `<script src="meu-script.js"></script>` (geralmente colocado antes do `</body>` para carregar o script após o HTML, ou no `<head>` usando atributo `defer` para atrasar a

execução até o DOM estar pronto). - **Embutido (Inline):** Inserir código entre tags `<script> ... </script>` diretamente no HTML. Exemplo:

```
<script>
  console.log("Olá do JavaScript!");
</script>
```

Isso pode ser útil para pequenos trechos ou exemplos, mas tal como CSS, separar em arquivo externo é melhor para organização e reuso.

Sintaxe Básica e Tipos: JavaScript tem uma sintaxe em C-like (parecida com Java/C/etc.): - **Instruções terminam com ;** (ponto-e-vírgula). Esse símbolo pode ser omitido em muitas situações, mas é recomendado usá-lo consistentemente para evitar ambiguidades. - **Comentários:** `// comentário` para linha única, `/* ... */` para bloco. - **Variáveis:** Em JS moderno, usamos `let` e `const` para declarar variáveis. `let` para variáveis mutáveis (valor pode mudar) e `const` para constantes (valor não muda após definido). Antigamente usava-se `var`, mas seu escopo funciona de maneira diferente (var é *escopo de função* ou global, enquanto let/const são *escopo de bloco* e não permitem redeclaração no mesmo escopo). **Use `let` e `const` preferencialmente**, pois evitam muitos problemas do `var`. - **Tipos primitivos:** número (*number*, que inclui inteiros e ponto flutuante), string (texto entre aspas simples `'...'` ou duplas `"..."` ou template strings com crases `...`), boolean (true/false), null (ausência deliberada de valor), undefined (valor não definido). Também existe *BigInt* (para números inteiros muito grandes) e *Symbol* (valor único). - **Tipos estruturados:** objetos e arrays. - *Array:* lista ordenada de valores, e.g., `let frutas = ["Maçã", "Banana", "Laranja];` (índices começam em 0). - *Objeto:* estrutura chave-valor, e.g., `let pessoa = { nome: "João", idade: 30 };`. Acesso por `pessoa.nome` ou `pessoa["idade"]`. - **Operadores:** Atribuição `=`, aritméticos `+ - * / %`, comparação `==` (igualdade frouxa, com coerção de tipo), `===` (estritamente igual, sem conversão de tipo - *prefira usar este*), `!=` e `!==` (diferente, estrito diferente), `>` `<` etc. Operadores lógicos: `&&` (E), `||` (OU), `!` (não). - **Coerção de tipos:** JS faz conversões de tipo automáticas em algumas comparações e operações, o que pode levar a surpresas, por isso prefira `===` sobre `==`. Por exemplo, `5 == "5"` é true (porque string "5" é convertida para número 5), mas `5 === "5"` é false (tipos diferentes, sem conversão) [10]. - **Estruturas de controle:** if/else, switch, loops (`for`, `while`, `do...while`). Exemplo:

```
if (idade >= 18) {
  console.log("Maior de idade");
} else {
  console.log("Menor de idade");
}
```

Laços:

```
for (let i = 0; i < 5; i++) {
  console.log("Iteração " + i);
}
```

ou `while(condição) { ... }`. Também `for...of` para iterar arrays e `for...in` para propriedades de objeto (menos usado).

Interação Básica com Usuário: Em ambiente de navegador, podemos usar: - `alert("Mensagem")` – mostra um popup de alerta. - `prompt("Pergunta")` – mostra um input popup e retorna o texto digitado pelo usuário (ou null se cancelado). - `console.log(valor)` – escreve no console do desenvolvedor (ótimo para debug). Exemplo simples:

```
<script>
  let nome = prompt("Qual é o seu nome?");
  alert("Olá, " + nome + "! Seja bem-vindo.");
</script>
```

Esse script perguntaria o nome e depois mostra um cumprimento. (Em muitos casos reais não se usam mais `alert/prompt` por não serem muito amigáveis, preferindo manipular o DOM para mostrar inputs e mensagens, mas para testes rápidos servem.)

Manipulando o DOM (Document Object Model): O DOM é a representação do HTML na forma de objeto em JS. Podemos acessar e modificar elementos: - `document.getElementById("id")` – retorna um elemento pelo id. - `document.querySelector("seletor")` – retorna o primeiro elemento que corresponda a um seletor CSS dado. - `document.querySelectorAll("seletor")` – retorna NodeList (lista de elementos) correspondentes. Uma vez com um elemento, podemos alterar seu conteúdo: `elemento.textContent = "novo texto";` ou `elemento.innerHTML = "<b>negrito</b>";`. Podemos mudar estilos: `elemento.style.color = "red";`, ou classes: `elemento.classList.add("ativa")`, etc.

Funções: Funções permitem reutilizar código. Declaração tradicional:

```
function soma(a, b) {
  return a + b;
}
let resultado = soma(2, 3); // 5
```

Também podemos armazenar funções em variáveis (funções anônimas ou *arrow functions*):

```
const soma2 = (a, b) => a + b; // arrow function, return implícito se uma
expressão
```

Acima, `soma2` faz o mesmo que `soma`. Arrow functions introduzidas no ES6 (2015) oferecem sintaxe curta e não criam seu próprio `this` (detalhe avançado). Para iniciantes: use a sintaxe que achar mais clara, sabendo que arrow é comum em código moderno, especialmente para callbacks.

Exemplo Simples de JS no DOM: Suponha que temos no HTML:

```
<button id="meuBotao">Clique aqui</button>
<p id="mensagem">Mensagem inicial.</p>
```

Podemos adicionar um script para reagir ao clique no botão e alterar o texto:

```
<script>
  const botao = document.getElementById("meuBotao");
  const paragrafo = document.getElementById("mensagem");
  botao.addEventListener("click", function() {
    paragrafo.textContent = "O botão foi clicado!";
  });
</script>
```

Comentário: Usamos `addEventListener` para configurar uma reação ao evento de clique no botão. Quando clicado, a função é executada e muda o conteúdo do parágrafo via `textContent`. Alternativamente, poderíamos no HTML pôr `onclick="..."` no botão, mas separar comportamento em JS é mais organizado.

Exercícios

1. **Cálculo Simples:** Escreva um script JS que pergunte dois números ao usuário (usando `prompt`), some-os e mostre o resultado em um `alert`. Certifique-se de converter as entradas de texto para número antes de somar.
2. **Sugestão de Solução:**

   ```
   let num1 = prompt("Digite o primeiro número:");
   let num2 = prompt("Digite o segundo número:");
   // Converte strings para número (parseFloat lida com decimais, parseInt
   para inteiros)
   let n1 = parseFloat(num1);
   let n2 = parseFloat(num2);
   let soma = n1 + n2;
   alert("A soma é: " + soma);
   ```

   Explicação: Usamos `parseFloat` para converter as strings retornadas pelo prompt em números (caso sejam valores não numéricos, o resultado será NaN – Not a Number). Somamos e exibimos. Se o usuário inserir, por exemplo, "5" e "7", o alert mostrará "A soma é: 12". Sem a conversão, as strings seriam concatenadas ("5" + "7" = "57"). Essa distinção é importante: em JS, operador `+` faz soma com números, mas concatena strings.
3. **Condicional:** Implemente um script que peça ao usuário um ano de nascimento e, ao clicar em um botão "Verificar", mostre no console se a pessoa é maior de idade ou não (considerando 18 anos). Dica: use `new Date().getFullYear()` para obter o ano atual.
4. **Sugestão de Solução (HTML + JS):**

   ```
   Ano de Nascimento: <input type="number" id="ano"><button
   id="verificar">Verificar</button>
   <script>
     const botao = document.getElementById("verificar");
     botao.addEventListener("click", function() {
       const anoNasc = Number(document.getElementById("ano").value);
       if (!anoNasc) {
         console.log("Por favor, insira um ano válido.");
         return;
       }
   ```

```
    const anoAtual = new Date().getFullYear();
    const idade = anoAtual - anoNasc;
    if (idade >= 18) {
      console.log("Maior de idade (idade: " + idade + ")");
    } else {
      console.log("Menor de idade (idade: " + idade + ")");
    }
  });
</script>
```

Explicação: Nesse código, pegamos o valor do input ano (convertido para Number). Verificamos se o valor é válido (se for 0 ou NaN, pedimos para inserir válido). Calculamos a idade aproximada subtraindo do ano atual. Então uma estrutura if/else determina a mensagem. O resultado é exibido no console; em uma página real, você poderia mostrar na tela, mas como não foi especificado elemento de saída, usamos console.log para conferir. Exemplo: se anoNasc = 2005 e anoAtual = 2025, idade = 20, o script escreve "Maior de idade (idade: 20)". Teste diferentes anos.

5. **Loop e Array:** Escreva uma função em JavaScript que receba um array de números e retorne o maior número encontrado. Por exemplo, dada a lista `[3, 7, 1, 9, 4]` deve retornar `9`. Não use funções prontas como `Math.max` ou `sort` - faça manualmente usando loop para prática.
6. **Sugestão de Solução:**

```
function encontrarMaior(nums) {
  if (nums.length === 0) {
    return null; // ou undefined, se array vazio
  }
  let maior = nums[0];
  for (let i = 1; i < nums.length; i++) {
    if (nums[i] > maior) {
      maior = nums[i];
    }
  }
  return maior;
}

console.log(encontrarMaior([3, 7, 1, 9, 4])); // Deve exibir 9
console.log(encontrarMaior([-10, -5, -20])); // -5
```

Explicação: A função `encontrarMaior` inicializa a variável `maior` com o primeiro elemento do array. Então itera pelos demais; se encontrar um elemento maior que o atual `maior`, atualiza. No final, `maior` contém o valor máximo. Testamos com um array de exemplo. Também testamos um caso com números negativos para garantir que funciona em qualquer situação. Esse tipo de lógica reforça compreensão de loops e condicionais. (Nota: Em entrevistas, é comum pedirem algo assim para ver o raciocínio, mesmo existindo soluções prontas).

JavaScript Intermediário

Estruturas de Dados e Métodos Utilitários: - *Arrays:* Possuem vários métodos úteis: `push()` (adiciona ao final), `pop()` (remove do final), `shift()` (remove do início), `unshift()` (adiciona no início), `length` (tamanho), `indexOf()` (encontra índice de um valor), `slice()` (extrai uma parte),

`splice()` (remove/insere elementos em posição arbitrária). E métodos modernos de iteração funcional: `forEach` (itera executando função), `map` (transforma cada elemento, retornando novo array), `filter` (filtra elementos baseados em condição), `reduce` (reduz o array a um único valor acumulando). - *Strings:* Também têm métodos: `.length`, `.toUpperCase()`, `.toLowerCase()`, `.includes("substr")`, `.indexOf("x")`, `.slice(start, end)`, `.replace("antigo", "novo")`, etc. - *Objetos:* Conceito de chave-valor. Podemos iterar atributos de objeto com `for...in` ou obter listas com `Object.keys(obj)` e `Object.values(obj)`. Objetos podem conter métodos (funções como valores de propriedades). Ex:

```
let pessoa = { nome: "Ana", saudacao: function() { console.log("Oi, " +
this.nome); } };
pessoa.saudacao(); // "Oi, Ana"
```

O `this` refere-se ao próprio objeto nesse contexto.

Funções Avançadas: - *Parâmetros padrão:* Podemos definir valor default para parâmetros de função:

```
function saldar(nome = "Visitante") {
  console.log("Olá, " + nome);
}
saldar(); // Olá, Visitante
```

- *Funções anônimas e callbacks:* Funções podem ser passadas como argumento para outras funções. Exemplo:

```
let numeros = [1,2,3];
numeros.forEach(function(x) { console.log(x*x); }); // imprime 1,4,9
```

Aqui usamos uma função anônima dentro de forEach para imprimir o quadrado de cada número. - *Escopo:* Em JS, escopos são por função ou por bloco (para let/const). Uma variável definida dentro de uma função não é acessível fora dela. Variáveis globais (declaradas fora de funções) podem ser acessadas em qualquer lugar (mas evite muitas globais para não causar conflitos). - *Closures:* Conceito importante: uma função interna lembra do ambiente em que foi criada, mesmo se executada fora dele. Por exemplo:

```
function criarContador() {
  let cont = 0;
  return function() {
    cont++;
    return cont;
  };
}
let contador = criarContador();
console.log(contador()); // 1
console.log(contador()); // 2
```

A função retornada mantém acesso à variável `cont` de `criarContador`, mesmo após esta ter terminado – isso é um *closure*. É avançado, mas útil para entender como funções guardam estado sem usar variáveis globais.

Manipulação do DOM (Intermediário): Além de ler/alterar texto e HTML, podemos criar e remover elementos:

```
let novoDiv = document.createElement("div");
novoDiv.textContent = "Olá!";
document.body.appendChild(novoDiv);
```

Isso cria um `<div>` com texto "Olá!" e adiciona no final do body. Podemos remover: `elemento.remove()` remove um nó. Também controlar atributos/propriedades: `elemento.src = "nova.png"; elemento.setAttribute("alt","nova imagem");`.

Eventos: Mais sobre eventos: além de click, temos eventos de teclado (`keydown`, `keyup`), de formulário (`submit` em forms), de mouse (`mouseover`, `mouseout` etc.), de página (`DOMContentLoaded` quando HTML carrega, `load` quando tudo carrega incluindo imagens). Podemos atribuir via `addEventListener` múltiplos handlers, ou sobrescrever usando propriedades `elemento.onclick = func`. Preferível addEventListener para poder adicionar vários e remover se necessário (`removeEventListener`).

JSON: Formato de dados *JavaScript Object Notation*. É basicamente texto formatado como objetos/arrays, muito usado para enviar/receber dados em APIs. Em JS, podemos converter: - De objeto JS para string JSON: `JSON.stringify(obj)`. - De string JSON para objeto JS: `JSON.parse(jsonString)`. Exemplo:

```
let obj = {nome: "Maria", idade: 25};
let json = JSON.stringify(obj);
// json = '{"nome":"Maria","idade":25}'
let obj2 = JSON.parse(json);
// obj2 agora é um objeto igual a obj
```

Conhecer JSON é importante pois em aplicações web realistas, dados entre front-end e back-end são trafegados nesse formato.

Boa Prática: Separar código: Em projetos, mantenha seu código JS modularizado. Evite misturar muita lógica no HTML inline. Use padrões como IIFE (função imediatamente invocada) para isolar escopo, ou módulos (se usando ES6 modules via `import/export`, que requer script type=module ou bundlers). Isso impede poluir o escopo global.

Pequenas funcionalidades úteis do ES6+: - *Template literals:* Strings com crase permitem interpolação:

```
let nome = "João";
console.log(`Olá, ${nome}!`);
```

Isso substitui `${nome}` pela variável, e permite multi-linhas sem concatenação. - *Desestruturação:* Extrair valores de array/objeto facilmente:

```
const [a, b] = [10, 20]; // a=10, b=20
const {nome, idade} = pessoa; // extrai propriedades para variáveis
```

- *Spread e Rest:* Spread `...` para espalhar elementos:

```
let arr1 = [1,2], arr2 = [3,4];
let combinado = [...arr1, ...arr2]; // [1,2,3,4]
```

Rest `...` em parâmetros de função para pegar vários argumentos em um array:

```
function somaTudo(...nums) {
  return nums.reduce((acc,n)=>acc+n, 0);
}
somaTudo(1,2,3,4); // 10
```

Exemplo Intermediário: Um pequeno programa que muda o estilo de um elemento quando um botão é clicado:

```
<button id="modo">Modo escuro</button>
<div id="conteudo">Texto de exemplo</div>
<script>
  const botao = document.getElementById("modo");
  const div = document.getElementById("conteudo");
  let escuro = false;
  botao.addEventListener("click", function() {
    if (!escuro) {
      document.body.style.backgroundColor = "#333";
      document.body.style.color = "#fff";
      botao.textContent = "Modo claro";
      escuro = true;
    } else {
      document.body.style.backgroundColor = "";
      document.body.style.color = "";
      botao.textContent = "Modo escuro";
      escuro = false;
    }
  });
</script>
```

Comentário: Esse script rudimentar implementa um alternador de tema claro/escuro. Ao clicar no botão, se estiver no modo claro, muda fundo/cor do body para tons escuros e altera texto do botão. Se clicar novamente, restaura (aqui usei `""` para backgroundColor e color para remover o estilo inline e voltar ao padrão CSS). Em uma aplicação real, seria melhor alternar uma classe no body (como `.dark-mode`)

e ter estilos CSS definidos para isso, mas o exemplo ilustra manipulação de estilo via JS e uso de variável de estado (`escuro`).

Exercícios

1. **Manipulação de Lista:** Suponha que você tem um HTML simples com uma lista `<ul id="lista"></ul>` e um campo de input e botão para adicionar tarefas. Implemente a função em JavaScript que pega o valor digitado no input e, ao clicar "Adicionar", insere esse texto como um novo `<li>` dentro da `<ul>`. Após adicionar, limpe o campo de input.
2. **Sugestão de Solução:** HTML:

   ```
   <input type="text" id="novoItem">
   <button id="addBtn">Adicionar</button>
   <ul id="lista"></ul>
   ```

   JavaScript:

   ```
   const input = document.getElementById("novoItem");
   const botao = document.getElementById("addBtn");
   const lista = document.getElementById("lista");
   botao.addEventListener("click", function() {
     const texto = input.value.trim();
     if (texto !== "") {
       const li = document.createElement("li");
       li.textContent = texto;
       lista.appendChild(li);
       input.value = ""; // limpa o campo
     }
   });
   ```

   Explicação: Selecionamos os elementos. No clique do botão, pegamos o valor do input e usamos `.trim()` para remover espaços extras (evitar entradas vazias ou só com espaço). Se houver texto, criamos um novo `<li>` via `document.createElement`, definimos seu `textContent` para o texto capturado e anexamos no `<ul>`. Depois limpamos o input para pronto para próxima entrada. Esse é um mini exemplo de *to-do list*. Para melhorar, poderíamos também acionar com Enter no input (ouvindo evento `keydown` para código 13), mas o enunciado focou no clique.
3. **Calculadora Simples:** Crie uma pequena "calculadora" web que tem dois campos numéricos e um `<select>` com a operação (`+`, `-`, `*`, `/`) e, ao clicar em um botão "Calcular", mostra o resultado em um elemento `<span id="resultado"></span>`. Implemente o cálculo em JavaScript.
4. **Sugestão de Solução:** HTML (exemplo):

   ```
   Número 1: <input type="number" id="n1">
   Operação: <select id="op">
     <option value="+">+</option>
     <option value="-">-</option>
     <option value="*">*</option>
     <option value="/">/</option>
   ```

```
</select>
Número 2: <input type="number" id="n2">
<button id="calcBtn">Calcular</button>
<p>Resultado: <span id="resultado"></span></p>
```

JS:

```
const n1 = document.getElementById("n1");
const n2 = document.getElementById("n2");
const op = document.getElementById("op");
const resultadoSpan = document.getElementById("resultado");
document.getElementById("calcBtn").addEventListener("click", function()
{
  const valor1 = parseFloat(n1.value);
  const valor2 = parseFloat(n2.value);
  const operacao = op.value;
  let resultado;
  if (isNaN(valor1) || isNaN(valor2)) {
    resultado = "Valores inválidos!";
  } else {
    switch (operacao) {
      case "+": resultado = valor1 + valor2; break;
      case "-": resultado = valor1 - valor2; break;
      case "*": resultado = valor1 * valor2; break;
      case "/":
        resultado = valor2 !== 0 ? (valor1 / valor2) : "Erro (divisão
por zero)";
        break;
    }
  }
  resultadoSpan.textContent = resultado;
});
```

Explicação: Ao clicar em calcular, pegamos os números dos inputs usando `parseFloat` (pois são do tipo string da interface). Verificamos se algum é NaN (not a number, caso o campo esteja vazio ou com valor inválido) e tratamos. Usamos um `switch` baseado na operação selecionada (`op.value` corresponde ao `value` da `<option>` escolhida). Realizamos a operação e guardamos em `resultado`. Para divisão, checamos divisão por zero para evitar Infinity. Por fim, exibimos no `resultadoSpan`. Essa lógica poderia ser extensível para mais operações facilmente. Atenção: `input type="number"` facilita pois impede entradas não numéricas em geral, mas ainda assim `value` vem como string, então precisa conversão. Isso demonstra manipulação de form e lógica básica em JS.

5. **Funções de Alta Ordem:** Dado um array de números em JavaScript, por exemplo `let dados = [1, 2, 3, 4, 5]`, use os métodos `map` e `filter` para primeiro criar um novo array com cada número ao quadrado e depois filtrar apenas os que forem maiores que 10. Mostre o resultado final no console.
6. **Sugestão de Solução:**

```
let dados = [1, 2, 3, 4, 5];
let aoQuadrado = dados.map(num => num * num);
let filtrados = aoQuadrado.filter(num => num > 10);
console.log("Original:", dados);
console.log("Quadrados:", aoQuadrado);
console.log("Filtrados (>10):", filtrados);
// Saída esperada:
// Original: [1, 2, 3, 4, 5]
// Quadrados: [1, 4, 9, 16, 25]
// Filtrados (>10): [16, 25]
```

Explicação: Aqui, utilizamos **funções de alta ordem** `map` e `filter`. `map` recebe uma função (arrow function no caso) que retorna o novo valor para cada elemento – multiplicamos o número por ele mesmo para obter quadrado. Isso resulta em `[1,4,9,16,25]`. Em seguida, `filter` recebe outra função que retorna true/false indicando se o elemento deve permanecer. Colocamos a condição `num > 10`, então ele vai manter apenas valores maiores que 10, resultando `[16, 25]`. Imprimimos para verificar. Esse tipo de operação declarativa (dizer *o quê* fazer em vez de usar loop explícito) é conciso e menos propenso a erros e muito utilizado no desenvolvimento moderno com JS (por exemplo, em programação funcional e bibliotecas como React para gerar listas de componentes).

JavaScript Avançado

Programação Assíncrona: JavaScript no navegador é single-thread (um único encadeamento de execução principal). Para tarefas que demoram (requisições de rede, temporizadores, leitura de arquivos, etc.), ele usa um modelo assíncrono com *callback queue* e *event loop* para não travar a interface. Conceitos importantes: - **setTimeout e setInterval:** `setTimeout(func, ms)` executa `func` uma vez após ms milissegundos. `setInterval(func, ms)` executa repetidamente a cada ms (até ser cancelado com `clearInterval`). Ex:

```
setTimeout(() => { console.log("Passaram 2 segundos"); }, 2000);
```

- **Callbacks:** Funções passadas como argumentos para serem chamadas quando uma operação assíncrona conclui. Ex:

```
function carregarDados(callback) {
  // simular demora
  setTimeout(() => {
    let dados = { user: "João" };
    callback(dados);
  }, 1000);
}
carregarDados(dados => {
  console.log("Dados recebidos:", dados);
});
```

Esse aninhamento de callbacks pode ficar complicado se há muitas sequências (callback hell). Para isso, vieram as Promises. - **Promises:** Introduzidas para melhorar a legibilidade de código assíncrono. Uma

Promise representa um resultado eventual de uma operação assíncrona: pode ser **resolvida** (cumprida) ou **rejeitada** (erro). Exemplo de uso:

```
function esperar(ms) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(), ms);
  });
}
esperar(1000).then(() => console.log("1 segundo passou"));
```

Aqui `esperar` retorna uma Promise que será resolvida após 1s, então usamos `.then` para definir o que fazer após a resolução. Um caso típico: funções assíncronas modernas como `fetch` (requisição HTTP) retornam Promise. Ex:

```
fetch('https://api.exemplo.com/dados')
  .then(response => response.json())
  .then(data => {
     console.log("Dados obtidos:", data);
  })
  .catch(error => {
     console.error("Erro na requisição:", error);
  });
```

O `.catch` captura rejeição (erros). Encadeamos vários then se precisarmos processar em etapas. - **Async/Await:** Sintaxe do ES2017 que permite escrever código assíncrono de forma mais linear, parecendo síncrona. Palavras-chave: - `async` antes de uma função a faz retornar uma Promise automaticamente, e permite usar `await` dentro dela. - `await` só pode ser usado dentro de funções async, e faz a função pausar até a Promise ser resolvida (ou rejeitada). Exemplo reescrevendo o fetch acima:

```
async function obterDados() {
  try {
    let response = await fetch('https://api.exemplo.com/dados');
    let data = await response.json();
    console.log("Dados obtidos:", data);
  } catch (error) {
    console.error("Erro na requisição:", error);
  }
}
obterDados();
```

Isso evita múltiplos `.then` aninhados e trata erros com try/catch de forma legível. A vantagem é estruturar o código como se fosse sequencial, mas sem bloquear a thread (de fato, a função assíncrona devolve o controle ao loop de eventos nas esperas).

Orientação a Objetos em JS: JS tem um modelo de protótipos. Cada objeto pode herdar de um protótipo (outro objeto). Tradicionalmente, criava-se "classes" usando funções construtoras e atribuía métodos no protótipo:

```
function Pessoa(nome) {
  this.nome = nome;
}
Pessoa.prototype.falar = function() {
  console.log("Olá, eu sou " + this.nome);
};
let p = new Pessoa("Maria");
p.falar(); // Olá, eu sou Maria
```

Desde ES6, temos sintaxe de classe que abstrage isso:

```
class Pessoa2 {
  constructor(nome) {
    this.nome = nome;
  }
  falar() {
    console.log(`Olá, eu sou ${this.nome}`);
  }
}
let q = new Pessoa2("José");
q.falar();
```

Por baixo dos panos, o funcionamento é prototípico (Pessoa2.prototype tem o método falar). Mas a sintaxe de classe suporta herança clara:

```
class Animal {
  constructor(nome) { this.nome = nome; }
  mover() { console.log(`${this.nome} se moveu.`); }
}
class Cachorro extends Animal {
  latir() { console.log("Au au!"); }
}
let rex = new Cachorro("Rex");
rex.mover(); // Rex se moveu.
rex.latir(); // Au au!
```

Aqui *extends* define herança (Cachorro herda de Animal). Podemos sobrescrever métodos e usar `super` para chamar o da classe base.

O operador `this`: `this` em JS é determinado pelo contexto de chamada: - Dentro de um método (função de um objeto chamada como objeto.metodo()), `this` refere-se ao objeto à esquerda do ponto. - Sozinho no contexto global, `this` refere-se ao objeto global (window no navegador), mas em modo estrito (strict mode), `this` global é undefined se fora de qualquer objeto/função. - Em eventos, `this` dentro de um handler atribuído via element.onclick será o elemento HTML (no addEventListener, dentro da função callback normal não atrelada, this = element também; mas em arrow functions, `this` não varia, ele mantém o do contexto léxico). - Para não se confundir, muitas vezes se usa arrow functions (que não têm this próprio) ou `.bind(this)` para fixar o valor do this quando necessário.

Gestão de Memória: JS tem coleta de lixo automática, então normalmente não precisamos liberar memória manualmente. Apenas devemos ter cuidado com referências (por ex., não criando referências circulares sem necessidade, ou não vazando variáveis no escopo global, pois essas só são liberadas ao fechar a página).

Ferramentas e Debugging: O console do navegador é seu amigo. Além de `console.log`, há `console.error`, `console.table` (visualiza arrays/objetos tabularmente), e no devtools podemos inspecionar objetos, usar breakpoints no código (painel "Sources"), ver chamadas de rede (painel "Network") e performance. Aprender a depurar é crucial: em JS, erros de sintaxe ou execução aparecerão no console (com linha e arquivo). Use o debugger do devtools ou insira `debugger;` no código JS para fazer o navegador pausar naquele ponto se estiver com devtools aberto.

Exemplo Avançado (Fetch API): Este exemplo mostra como consumir uma API pública de maneira simples. Vamos supor que existe uma API que retorna dados de usuários em JSON:

```
async function carregarUsuarios() {
  try {
    const resp = await fetch('https://jsonplaceholder.typicode.com/users');
    if (!resp.ok) { // resp.status para código
      throw new Error("Erro na resposta: " + resp.status);
    }
    const usuarios = await resp.json();
    usuarios.forEach(u => {
      console.log(`Nome: ${u.name} - Email: ${u.email}`);
    });
  } catch (e) {
    console.error("Falha ao carregar usuários:", e);
  }
}
carregarUsuarios();
```

Comentário: Aqui usamos `fetch` para GET de uma URL (jsonplaceholder.typicode.com/users é uma fake API que retorna lista de usuários). Verificamos `resp.ok` (que indica status 200-299). Se ok, extraímos JSON. Então iteramos e logamos nome e email de cada usuário. Em caso de erro (ex: rede offline ou status errado), capturamos no catch e mostramos no console. Notar o uso de `forEach` e template string. Esse padrão de fetch + async/await é muito usado para buscar dados de servidor.

Nota sobre Segurança: Quando usar JS no front-end, evite práticas inseguras: - Não use `eval` para executar strings como código, isso abre brecha XSS. - Quando inserir conteúdo vindo de usuário no DOM, sanitize ou use textContent (que não interpreta HTML) para prevenir injeção de HTML/script. - Ative o modo estrito (`"use strict";` no topo do script ou função) para ter verificações mais rígidas (evitar variáveis globais acidentais, etc). - Cuidado ao expor dados sensíveis; lembre-se que JS do front-end é visível ao usuário (não coloque chaves secretas lá, por exemplo).

Exercícios

1. **Promessas e async/await:** Implemente uma função que retorna uma Promise que se resolve após 3 segundos retornando o texto `"Concluído"`. Em seguida, mostre duas maneiras de utilizar essa função: a) com `.then()` e b) com `async/await`.

2. **Sugestão de Solução:**

```
function demoraTresSegundos() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("Concluído");
    }, 3000);
  });
}

// a) Usando .then()
demoraTresSegundos().then(mensagem => {
  console.log("Promise via .then: " + mensagem);
});

// b) Usando async/await
async function executar() {
  console.log("Esperando 3 segundos...");
  const msg = await demoraTresSegundos();
  console.log("Promise via async/await: " + msg);
}
executar();
```

Explicação: A função `demoraTresSegundos` cria uma nova Promise que apenas espera 3000 ms e então chama `resolve("Concluído")`. No uso a), encadeamos `.then` para tratar quando a promise for resolvida, imprimindo a mensagem. No b), uma função async `executar` usa `await` para pausar até obter o resultado da promise, depois loga. O console deverá mostrar (após 3 segundos):

```
Esperando 3 segundos...
Promise via .then: Concluído
Promise via async/await: Concluído
```

A ordem entre as duas mensagens de "Concluído" pode variar dependendo do timing, mas ambas chegam ~3s depois. Isso demonstra as duas formas equivalentes, sendo a de async/await geralmente mais legível quando temos várias operações sequenciais.

3. **Manipulação de Classes via JS:** Suponha que você tenha um elemento `<div id="box" class="verde"></div>` representando um quadrado verde, com estilos CSS pré-definidos para classes .verde (fundo verde) e .vermelho (fundo vermelho). Crie um script que, a cada clique no `div`, alterna sua cor: se estiver verde, muda para vermelho; se estiver vermelho, volta para verde. Use a propriedade `classList` do elemento.
4. **Sugestão de Solução:**

```
const box = document.getElementById("box");
box.addEventListener("click", function() {
  if (box.classList.contains("verde")) {
    box.classList.remove("verde");
    box.classList.add("vermelho");
  } else {
```

```
    box.classList.remove("vermelho");
    box.classList.add("verde");
  }
});
```

Explicação: Selecionamos o elemento com id "box". No evento de clique, usamos `classList.contains` para verificar se ele tem a classe "verde". Caso tenha, removemos "verde" e adicionamos "vermelho". Caso contrário (presumivelmente está vermelho), removemos "vermelho" e botamos "verde". Assim alternamos a classe. Outra forma mais curta: `box.classList.toggle("verde")` e `box.classList.toggle("vermelho")`, mas isso poderia levar a dois ao mesmo tempo se não planejar certo – melhor explicitamente como acima. Ao usar classes definidas no CSS, se o CSS contém `.verde { background: green; } .vermelho { background: red; }`, o quadrado muda de cor conforme a classe sem precisar mudar estilos manualmente via JS, o que é uma boa separação de responsabilidades.

5. **Quiz de Conhecimento (OOP):** Crie uma classe `Livro` em JavaScript ES6 que tenha propriedades `titulo`, `autor` e `ano`. Adicione um método `getResumo()` que retorna uma string resumindo as informações do livro. Em seguida, crie uma subclasse `Ebook` que estende `Livro` e adiciona a propriedade `formato` (por exemplo: "PDF", "EPUB"). Override o método `getResumo()` na classe `Ebook` para incluir o formato no resumo. Mostre exemplo de criação de instâncias e uso dos métodos no console.
6. **Sugestão de Solução:**

```
class Livro {
  constructor(titulo, autor, ano) {
    this.titulo = titulo;
    this.autor = autor;
    this.ano = ano;
  }
  getResumo() {
    return `${this.titulo} (${this.ano}), escrito por ${this.autor}.`;
  }
}

class Ebook extends Livro {
  constructor(titulo, autor, ano, formato) {
    super(titulo, autor, ano);
    this.formato = formato;
  }
  getResumo() {
    return `${this.titulo} (${this.ano}), ${this.formato} - autor: $
{this.autor}.`;
  }
}

// Testando:
const livroFisico = new Livro("Dom Casmurro", "Machado de Assis", 1899);
const livroDigital = new Ebook("JavaScript Guia", "Fulano", 2020,
"PDF");
```

```
console.log(livroFisico.getResumo());
console.log(livroDigital.getResumo());
```

Explicação: A classe `Livro` define as propriedades básicas e um método que usa template strings para retornar "Título (Ano), escrito por Autor." A classe `Ebook` estende `Livro` usando `extends`. No construtor de Ebook, chamamos `super(...)` para aproveitar a inicialização de título, autor, ano na classe base, e depois definimos `this.formato`. Em `getResumo()` do Ebook, sobrescrevemos a versão pai: incluímos o formato na string de resumo. Criamos uma instância de Livro comum e uma de Ebook. Ao chamar seus getResumo, o primeiro utiliza a versão de Livro, o segundo usa a versão override de Ebook. Por exemplo, a saída seria:

```
Dom Casmurro (1899), escrito por Machado de Assis.
JavaScript Guia (2020), PDF - autor: Fulano.
```

Isso demonstra conceitos de POO: classes, herança, super chamada, override de métodos e uso de instâncias. É válido notar que em JS poderíamos fazer sem classes usando protótipos diretamente ou até funções fábrica, mas a sintaxe de classe deixa mais claro.

PHP

PHP Iniciante

O que é PHP: PHP (um acrônimo recursivo para "PHP: Hypertext Preprocessor") é uma linguagem de script interpretada do lado do servidor, muito utilizada para desenvolvimento web dinâmico. Criada originalmente em 1994, evoluiu bastante desde então e hoje é uma linguagem orientada a objetos e com ampla funcionalidade. Em páginas web, o PHP é executado no servidor gerando HTML (ou outros dados) que são enviados ao cliente. Por isso, ao contrário do JavaScript, o PHP não é visto nem executado no navegador; ele roda no servidor e produz a resposta para o browser.

Sintaxe Básica e Embutindo em HTML: Um arquivo PHP geralmente tem extensão `.php`. Podemos escrever HTML normalmente e inserir trechos de código PHP usando as tags especiais:

```
<?php ... código PHP ... ?>
```

Tudo dentro dessas tags será interpretado pelo servidor como código PHP; o que estiver fora é enviado diretamente (ex: HTML puro). Exemplo mínimo:

```
<!DOCTYPE html>
<html>
<body>
  <h1><?php echo "Olá, mundo do PHP!"; ?></h1>
</body>
</html>
```

Aqui, o `<?php echo "Olá, mundo do PHP!"; ?>` fará o PHP imprimir "Olá, mundo do PHP!" dentro do `<h1>`. O comando `echo` serve para enviar saída (nesse caso, texto) ao cliente. A tag de fechamento `?>` pode ser omitida se o arquivo terminar com PHP, para evitar problemas de whitespace [9], mas em meio ao HTML usamos abertura e fechamento conforme necessário.

Variáveis e Tipos: Em PHP, todas as variáveis começam com `$`. Ex: `$nome = "João"; $idade = 20;`. PHP é de *tipagem dinâmica*, ou seja, você não declara tipo, ele é determinado em runtime. Principais tipos: - **String:** Texto, entre aspas simples `'` ou duplas `"`. Aspas duplas interpretam variáveis e sequências de escape; aspas simples não. Ex: `$s1 = "Oi $nome"; $s2 = 'Oi $nome';` => $s1 substitui $nome, $s2 não. - **Integer:** Número inteiro (32 ou 64 bits dependendo do sistema). - **Float (double):** Número de ponto flutuante (decimal). - **Boolean:** true ou false. - **Array:** Estrutura de vetor/mapa. Em PHP arrays podem ser numéricos (lista indexada por números) ou associativos (chaves nomeadas), ou mistos. - **Null:** valor nulo (ausência de valor). Atribui com `$var = null;`.

Exemplo:

```
<?php
  $produto = "Bicicleta";
  $preco = 1200.50;
  $emEstoque = true;
?>
```

Além desses, existem objetos (instâncias de classes), recursos (referências a recursos externos, ex: conexão de banco), etc.

Operadores e Sintaxe de Controle: - Operadores aritméticos: `+ - * / %`. - Concatenação de strings: o operador `.` (ponto) concatena strings. Ex: `$nomeCompleto = $nome . " " . $sobrenome;`. - Atribuição: `=`, e incrementais: `+=, -=, .=`, etc. - Comparação: `==` (igualdade de valor com conversão de tipo se necessário), `===` (identidade, valor e tipo iguais) [10], `!=` ou `<>` (diferente), `!==` (não idêntico). Tenha cuidado: em PHP, `"123" == 123` é true (porque converte e compara valor), mas `"123" === 123` é false (tipos diferentes, string vs int) [10]. Também, `==` de forma frouxa pode ter surpresas (por exemplo, 0 == "abc" resulta true, pois "abc" vira 0 na conversão). Portanto, **prefira `===` e `!==` para comparações previsíveis** [10]. - Lógicos: `&&` (E), `||` (OU), `!` (não). Também `and`, `or` com precedência diferente (cuidado ao misturar). - if/else:

```
if ($idade >= 18) {
  echo "Maior de idade";
} elseif ($idade >= 16) {
  echo "Menor de idade, mas já pode votar!";
} else {
  echo "Menor de idade";
}
```

- switch/case, semelhante a C/JS:

```
switch($op) {
  case "+":
    $res = $a + $b;
    break;
  case "-":
    $res = $a - $b;
    break;
  default:
```

```
    echo "Operação inválida";
}
```

- Loops: `for`, `while`, `do...while` são similares a C/JS. O foreach é muito usado para arrays:

```
$numeros = [10, 20, 30];
foreach ($numeros as $num) {
  echo $num;
}
```

Ou para associativos:

```
$idades = ["Ana"=>25, "Bruno"=>32];
foreach ($idades as $nome => $idade) {
  echo "$nome tem $idade anos.";
}
```

Funções Simples e Escopo: Definimos com `function nomeDaFunc($param1, $param2=valorPadrao) { ... }`. Exemplo:

```
<?php
  function soma($x, $y = 0) {
    return $x + $y;
  }
  echo soma(3,4); // 7
  echo soma(5);   // 5 (usou $y padrão 0)
?>
```

Em PHP, variáveis definidas fora de funções são globais e não acessíveis dentro da função a menos que use `global $var;` ou passe por parâmetro. Escopo local é por função.

Superglobais: PHP fornece variáveis superglobais acessíveis de qualquer lugar: - `$_GET`, `$_POST`: arrays associativos com dados enviados por HTTP GET ou POST (formularios, query strings). - `$_SERVER`: info do servidor e requisição (ex: `$_SERVER['REQUEST_METHOD']`, `$_SERVER['PHP_SELF']`). - `$_SESSION`, `$_COOKIE`: dados de sessão e cookies. - `$_FILES`: arquivos enviados via upload. - `$_REQUEST`: combina GET/POST/COOKIE (geralmente não usar por segurança, prefira específico). - `$_ENV`: variáveis de ambiente.

Exemplo: se um formulário faz GET para esta página com ?nome=Ana,

```
$nome = $_GET['nome'];
echo "Olá, $nome";
```

isso pegaria o valor. Sempre valide/sanitize esses inputs em real, mas isso é fundamental para capturar dados de formulários.

Formulário -> PHP (breve fluxo): No HTML:

```
<form action="processa.php" method="POST">
  Nome: <input name="nome">
  <button type="submit">Enviar</button>
</form>
```

No processa.php:

```
<?php
  $nome = $_POST['nome'];
  echo "Recebido nome: " . htmlspecialchars($nome);
?>
```

Note o uso de `htmlspecialchars` – converte caracteres especiais em entidades HTML, prevenindo XSS se alguém enviar `<script>` no campo, por exemplo. Sempre que ecoar dados do usuário, é boa prática sanitar ou escapar para evitar injeções de HTML/JS.

Exemplo Simples: Um pequeno script PHP autônomo:

```
<?php
  $a = 5;
  $b = 7;
  $soma = $a + $b;
  echo "<p>A soma de $a e $b é $soma.</p>";
?>
```

Isso imprimiria: "A soma de 5 e 7 é 12." Observa que as variáveis dentro da string de aspas duplas foram interpoladas. Poderia também concatenar: `echo "<p>A soma de ".$a." e ".$b." é ". $soma."</p>";`.

Incluindo outros arquivos: Em PHP, para organizar, usamos `include` ou `require` para inserir código de outros arquivos. Por exemplo, `include 'header.php';` vai literalmente trazer o conteúdo daquele arquivo. A diferença é: - `include`: em caso de falha (arquivo não encontrado), emite um *warning* e o script continua tentando rodar. - `require`: em caso de falha, lança um *erro fatal* e para a execução [11]. Também existem `include_once` e `require_once` que garantem que o arquivo seja incluído apenas uma vez (evita duplicata de funções ou reexecução acidental). Use require para arquivos críticos (como configurações, classes essenciais) e include para opcionais. A versão _once para evitar problemas de múltipla inclusão.

Saída e cabeçalhos: Usamos `echo` para saída básica. Também `print` (similar, mas é uma expressão com retorno), e funções como `printf` (formatação estilo C), `var_dump` ou `print_r` para debug (mostrar estrutura de variáveis). Para enviar cabeçalhos HTTP (redirecionar, ou definir content-type, cookies), usamos `header()` antes de ter enviado qualquer output. Ex: `header("Location: obrigado.html"); exit;` redireciona e termina o script. Ou `header("Content-Type: application/json"); echo json_encode($data);` para retornar JSON.

Exercícios

1. **Saída Simples:** Crie um script PHP que defina duas variáveis, `$nome` e `$hora` (esta última obtendo a hora atual com `date("H")` por exemplo), e use essas variáveis para imprimir uma saudação dinâmica. Por exemplo: "Bom dia, Maria!" ou "Boa noite, João!", dependendo da hora (considere <12 manhã, <18 tarde, senão noite).
2. **Sugestão de Solução:**

```
<?php
  date_default_timezone_set('America/Sao_Paulo'); // garante fuso
horário correto
  $nome = "Carlos";
  $hora = date("H"); // hora em formato 24h, string com 00-23
  $hora = (int)$hora; // converte para int
  if ($hora < 12) {
    $saudacao = "Bom dia";
  } elseif ($hora < 18) {
    $saudacao = "Boa tarde";
  } else {
    $saudacao = "Boa noite";
  }
  echo "<h3>$saudacao, $nome!</h3>";
?>
```

   Explicação: Usamos `date("H")` para pegar a hora atual do sistema (configuramos timezone para evitar discrepâncias). Convertido para inteiro. Usamos if/elseif para decidir a saudação. Depois usamos echo com a string interpolada em HTML. Se $nome = Carlos e hora atual 14, por exemplo, exibirá "Boa tarde, Carlos!".
3. **Formulário e Processamento:** Suponha um formulário HTML que envia via POST um campo "numero". Faça um script PHP (`processa.php`) que receba esse número, verifique se ele é par ou ímpar, e exiba uma mensagem adequada. Inclua tratamento caso nenhum número seja enviado (campo vazio).
4. **Sugestão de Solução (processa.php):**

```
<?php
  if (!isset($_POST['numero']) || $_POST['numero'] === '') {
    echo "Nenhum número foi fornecido.";
    exit;
  }
  $numStr = $_POST['numero'];
  // Podemos usar filtro para garantir que é número
  if (!is_numeric($numStr)) {
    echo "Valor inválido! Por favor, envie um número.";
    exit;
  }
  $num = (int)$numStr;
  if ($num % 2 == 0) {
    echo "O número $num é <strong>Par</strong>.";
  } else {
    echo "O número $num é <strong>Ímpar</strong>.";
```

```
  }
?>
```

Explicação: Primeiro, checamos com isset e `=== ''` se o campo veio vazio ou não veio, exibindo mensagem e terminando (`exit`) para não prosseguir. Depois usamos `is_numeric` para aceitar apenas valores numéricos (isso permite floats também; se quiséssemos só inteiro, poderíamos usar ctype_digit para dígitos). Convertendo para (int) coverte float para int truncando, mas aqui ok. Verificamos resto da divisão por 2 para determinar paridade. E mostramos com formatação simples em negrito. Esse script precisa ser alimentado por um form, ex:

```
<form action="processa.php" method="POST">
  <label>Digite um número:</label>
  <input type="text" name="numero">
  <button type="submit">Enviar</button>
</form>
```

5. **Array e Loop:** No PHP, crie um array `$frutas` com pelo menos 5 nomes de frutas. Em seguida, escreva um trecho que percorre esse array (use `foreach`) e exibe cada fruta em uma lista HTML `<ul><li>...`.
6. **Sugestão de Solução:**

```
<?php
  $frutas = ["Maçã", "Banana", "Laranja", "Melancia", "Uva"];
  echo "<ul>";
  foreach ($frutas as $fruta) {
    echo "<li>$fruta</li>";
  }
  echo "</ul>";
?>
```

Explicação: Criamos um array de strings. Iniciamos a lista com `<ul>`. O foreach itera cada valor (a variável $fruta dentro do loop assume cada valor do array sequencialmente). Dentro do loop, para cada fruta, imprimimos um `<li>Fruta</li>`. Após o loop, fechamos o `ul`. O resultado HTML será uma lista não ordenada com cada fruta em um item. Isso ilustra manipulação básica de array e output de conteúdo dinâmico. (Observação: poderia concatenar numa string e imprimir uma vez só, mas usar echo dentro do loop é comum e suficiente aqui.)

PHP Intermediário

Arrays Associativos e Multidimensionais: - Associativo: chave => valor. Ex:

```
$produto = [
  "nome" => "Camisa",
  "preco" => 79.90,
  "em_estoque" => true
];
echo $produto["nome"]; // Camisa
```

Acesso via chave string. Use `foreach ($produto as $chave => $valor)` para iterar pares chave-valor. - Multidimensional: array contendo arrays. Ex:

```
$matriz = [
  [1, 2, 3],
  [4, 5, 6]
];
echo $matriz[1][2]; // 6 (segunda linha, terceira coluna)
```

Ou array de assoc dentro de assoc etc. Pode aninhar livremente.

Funções de Arrays Úteis: PHP tem riquíssima biblioteca de arrays: - `count($arr)` conta elementos. - `array_push($arr, $valor)` adiciona no final (ou `$arr[] = $valor` sintaxe curta). - `array_pop($arr)` remove e retorna último. - `array_shift`, `array_unshift` (remove/adiciona no início). - `sort($arr)` ordena vetor (e `rsort` reverso). Para associativos, `asort` mantém associação ordenando pelos valores, `ksort` pelas chaves. - `in_array($valor, $arr)` verifica se valor existe. - `array_merge($a1, $a2)` junta arrays. - `explode($sep, $str)` divide string em array pelo separador. - `implode($sep, $arr)` junta array em string separada pelo sep. - `array_map($func, $arr)` aplica função em cada elemento, retornando novo array. - Entre muitas outras (filter, reduce no PHP chama array_filter, array_reduce, etc.).

Inclusão e Reutilização de Código: Como mencionado, temos `include/require`. Em aplicações, costumamos ter um arquivo de configuração (db, etc) que damos require no topo, e um arquivo de funções comuns. Use `require_once` para classes ou config para evitar duplicatas.

Programação Orientada a Objetos (POO) Básica: - Definir classes com `class Nome { ... }`. - Propriedades (atributos) e métodos (funções dentro da classe). - Visibilidade: `public` (acessível de fora), `private` (somente dentro da classe), `protected` (dentro da classe e de subclasses). - Construtor: método especial `__construct` chamado ao instanciar. - `$this` para acessar membros da instância dentro da classe. - Instanciar: `$obj = new NomeClasse(args);`. - Exemplo:

```
class Pessoa {
  public $nome;
  private $idade;
  public function __construct($nome, $idade) {
    $this->nome = $nome;
    $this->idade = $idade;
  }
  public function apresentar() {
    return "Olá, eu sou $this->nome e tenho $this->idade anos.";
  }
  public function getIdade() {
    return $this->idade; // acesso permitido internamente
  }
}
$p = new Pessoa("Alice", 30);
echo $p->apresentar();
```

Observação: No PHP, se não especificar, a visibilidade padrão de propriedades é `public` (mas é boa prática sempre declarar explicitamente). Métodos também default público se não indicado. Private evita acesso externo: por exemplo, `$p->idade` não seria acessível fora, precisa de um getter ou método público para expor se necessário. - Herança: `class Funcionario extends Pessoa { ... }`. PHP só suporta herança simples (uma única classe pai) [12] . Em uma subclasse, podemos chamar o construtor pai via `parent::__construct()` se sobrescrever. - PHP também suporta **interfaces** (declaração de métodos sem implementação para impor contratos) e **classes abstratas** (classe que não pode instanciar, serve de base, podendo ter métodos abstratos a implementar nas filhas). E **traits** (desde PHP 5.4) que permitem reutilizar conjuntos de métodos em múltiplas classes - uma forma de composição horizontal para contornar falta de herança múltipla [13] .

PDO e MySQL (banco de dados): Interagir com base de dados é comum. Em PHP moderno, usamos PDO (PHP Data Objects) ou MySQLi. *PDO* é uma interface unificada que suporta vários SGBDs (MySQL, PostgreSQL, etc). Exemplo de conexão MySQL com PDO:

```
try {
  $pdo = new PDO("mysql:host=localhost;dbname=testdb;charset=utf8",
"usuario", "senha");
  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  // Executar uma consulta
  $stmt = $pdo->query("SELECT * FROM produtos");
  $lista = $stmt->fetchAll(PDO::FETCH_ASSOC);
  foreach($lista as $produto) {
    echo $produto['nome'] . " - R$ " . $produto['preco'] . "<br>";
  }
} catch (PDOException $e) {
  echo "Erro na conexão ou consulta: " . $e->getMessage();
}
```

Aqui usamos try/catch para pegar PDOException. A conexão DSN indica driver mysql, host e dbname, e configuramos para lançar exceções em erros. `query` executa e retorna um statement, depois `fetchAll` para pegar todos resultados como array associativo. Também poderia usar `prepare` e `execute` com parâmetros para consultas seguras (ex: inserção com `:param` placeholders).

Sessões e Cookies: Sessões permitem manter dados do usuário entre requisições usando um identificador (geralmente cookie PHPSESSID). - Para usar sessões: chame `session_start();` no topo da página (antes de output) para iniciar/resumir a sessão. - Então use `$_SESSION` array superglobal para armazenar dados por sessão. Ex:

```
session_start();
$_SESSION['usuario'] = "Joao";
```

Em outra página, após session_start, `echo $_SESSION['usuario'];` terá "Joao". - Para destruir sessão: `session_destroy();` (apaga no server, mas cookie no cliente precisa expirar). - Cookies: `setcookie("nome", "valor", $expiraTimestamp, "/caminho", "dominio", $secure, $httponly);`. Isso envia header Set-Cookie. Depois, nas próximas requisições, o cookie vem em `$_COOKIE['nome']`.

Exemplo de Sessão rápida: Autenticação trivial:

```
// login.php
session_start();
if ($_POST['senha'] === '1234') {
  $_SESSION['auth'] = true;
  header("Location: restrita.php");
} else {
  echo "Senha incorreta";
}
```

```
// restrita.php
session_start();
if (!isset($_SESSION['auth']) || $_SESSION['auth'] !== true) {
  die("Acesso negado.");
}
echo "Conteúdo restrito aqui.";
```

Isso guarda um flag de autenticação na sessão.

Manipulação de Arquivos: PHP lida bem com arquivos: - `file_get_contents($filename)` lê todo conteúdo de um arquivo (ou URL se allow_url_fopen). - `file_put_contents($filename, $data)` escreve (sobrescreve) dados em arquivo (pode usar FILE_APPEND flag para anexar). - Funções clássicas: `fopen, fread, fwrite, fclose` para controle mais fino. - `unlink($filename)` exclui arquivo.

Erros e Exceções: Tipos de erro em PHP: - Notices: pequenos avisos (ex: usar variável indefinida gera notice). Não interrompem a execução. - Warnings: avisos mais sérios (ex: incluir arquivo inexistente por include gera warning). Também não param execução. - Fatal errors: erros graves (ex: chamar função não definida, erro de parse), terminam o script. - Exceções: lançadas manualmente com `throw new Exception("msg")` ou por algumas extensões (PDO no ERRMODE_EXCEPTION). Pode envolver com try/catch:

```
try {
  // ...
  throw new Exception("Algo deu errado");
} catch(Exception $e) {
  echo "Exceção capturada: " . $e->getMessage();
}
```

Se não capturada, exceção não tratada também interrompe script (fatal). - Config PHP (php.ini) define quais erros são exibidos ou apenas logados. Em desenvolvimento, usar `error_reporting(E_ALL);` `ini_set('display_errors', 1);` para ver todos.

Exemplo de Função e Uso de Biblioteca: Vamos supor que precisamos formatar uma data em português. Existe a função date() mas formatação de nome de mês em português envolve setlocale e strftime:

```
function formatarDataPt($dataStr) {
  setlocale(LC_TIME, 'pt_BR.UTF-8');
  $timestamp = strtotime($dataStr);
  if (!$timestamp) {
    return "Data inválida";
  }
  return strftime("%d de %B de %Y", $timestamp);
}
echo formatarDataPt("2025-06-07");
// Exemplo de saída: "07 de junho de 2025"
```

Aqui vemos uso de função customizada, `strtotime` para converter string para timestamp, e `strftime` com locale PT-BR para obter mês em português.

Exercícios

1. **Funções e Escopo:** Crie uma função `fatorial($n)` em PHP que calcule o fatorial de um número inteiro $n (não negativo). Use um loop ou recursão na implementação. Em seguida, escreva um código que lê um valor (pode simular atribuindo a uma variável) e exibe `"$n! = resultado"`.
2. **Sugestão de Solução:**

```
<?php
  function fatorial($n) {
    if ($n < 0) {
      return null; // fatorial não definido para negativos
    }
    $fat = 1;
    for ($i = 1; $i <= $n; $i++) {
      $fat *= $i;
    }
    return $fat;
  }

  $valor = 6;
  $resultado = fatorial($valor);
  if ($resultado !== null) {
    echo "$valor! = $resultado";
  } else {
    echo "Valor inválido para fatorial.";
  }
?>
```

Explicação: A função fatorial inicializa $fat=1 e multiplica de 1 até $n. Poderia usar recursão (`return $n<=1 ? 1 : $n * fatorial($n-1);`), mas com loop é simples. Tratamos $n negativo retornando null. Ao usar, checamos se retornou null para mensagem de erro. Para $valor=6, deve imprimir "6! = 720". Teste com 0 (0! =1) e 1 (1! =1) também. Esse exercício reforça função, loop e validação simples.

3. **CRUD Simulado de Array:** Imagine que você tem um array associativo `$clientes` onde as chaves são IDs e os valores são nomes de clientes. Ex:

```
$clientes = [
  101 => "Ana",
  102 => "Bruno",
  103 => "Carlos"
];
```

Escreva um código PHP para:

4. (a) Adicionar um novo cliente com ID 104 e nome "Denise".
5. (b) Atualizar o nome do cliente ID 102 para "Bruno Silva".
6. (c) Remover o cliente de ID 101.
7. (d) Exibir todos os clientes restantes no formato "ID: Nome".
8. **Sugestão de Solução:**

```
<?php
  $clientes = [
    101 => "Ana",
    102 => "Bruno",
    103 => "Carlos"
  ];
  // a) Adicionar
  $clientes[104] = "Denise";
  // b) Atualizar
  if (isset($clientes[102])) {
    $clientes[102] = "Bruno Silva";
  }
  // c) Remover
  unset($clientes[101]);
  // d) Exibir
  foreach ($clientes as $id => $nome) {
    echo "$id: $nome<br>";
  }
?>
```

Explicação: Para adicionar, basta atribuir `$clientes[104] = "Denise";` já que a chave não existia, cria nova entrada. Atualizar, checamos se existe id 102 (good practice) e sobrescrevemos valor. Remover, usamos `unset($clientes[101]);` que exclui aquele par do array. Por fim, percorremos com foreach para imprimir cada id e nome numa linha com `<br>`. Após essas operações, o array deve ter 102 => "Bruno Silva", 103 => "Carlos", 104 => "Denise". O loop exibirá:

```
102: Bruno Silva
103: Carlos
104: Denise
```

A ordem do foreach será pela ordem das chaves internas (inserção ou ordenação interna associativa não garantida a menos que useksort; mas aqui provavelmente aparece 102,103,104 nessa ordem).

9. **POO Básico:** Defina uma classe `Retangulo` em PHP com propriedades `largura` e `altura` (públicas). Forneça um método `area()` que calcula a área (largura * altura) e um método `perimetro()` que calcula o perímetro (2*(largura+altura)). Então, crie uma instância dessa classe, defina largura=5 e altura=3, e exiba os valores calculados de área e perímetro.
10. **Sugestão de Solução:**

```php
<?php
  class Retangulo {
    public $largura;
    public $altura;
    public function area() {
      return $this->largura * $this->altura;
    }
    public function perimetro() {
      return 2 * ($this->largura + $this->altura);
    }
  }

  $ret = new Retangulo();
  $ret->largura = 5;
  $ret->altura = 3;
  echo "Área: " . $ret->area() . " m^2<br>";
  echo "Perímetro: " . $ret->perimetro() . " m<br>";
?>
```

Explicação: A classe Retangulo tem atributos largura e altura. Criamos métodos area() e perimetro() usando `$this` para acessar os atributos do objeto instanciado. Note que não definimos __construct, poderíamos ter e passar os valores no new. Mas aqui setamos depois manualmente. Ao instanciar `$ret = new Retangulo();`, definimos as propriedades, então chamamos métodos e imprimimos. Para largura=5, altura=3, área = 15, perímetro = 16. Assim deve mostrar "Área: 15 m^2" e "Perímetro: 16 m". Isso revisa criação de classe e uso de métodos. Em um contexto web, esses valores poderiam vir de um formulário, etc., mas aqui fixamos para teste.

PHP Avançado

Tópicos Avançados e Boas Práticas:

- **Namespaces:** Em PHP, para organizar classes (especialmente em projetos grandes e pacotes), usam-se *namespaces*. Declarados no topo de arquivos PHP: `namespace MeuProjeto\Modulo;`. Isso evita conflitos de nomes de classes/funções entre bibliotecas diferentes. Para usar classes de um namespace, ou referenciamos com use: `use MeuProjeto\Modulo\ClasseX; $obj = new ClasseX();` ou usamos o FQCN (Fully Qualified Class Name) `new \MeuProjeto\Modulo\ClasseX();`. Namespaces são cruciais ao usar autoloaders e Composer (gestor de dependências).

- **Composer e Autoloading:** Composer é o gerenciador de pacotes do PHP. Ele permite declarar dependências (em arquivo composer.json) e instala bibliotecas (como PHPMailer, Guzzle HTTP client, etc.). Composer gera um *autoload* que carrega automaticamente as classes necessárias sem precisar de require manual se as classes seguem PSR-4 (padrão de autoloading de classes

por namespace e estrutura de diretórios). Em projetos modernos, você raramente usa require para suas classes; você configura autoload e simplesmente instância as classes (o autoloader irá encontrar e incluir o arquivo correspondente).

- **PSR (PHP Standards Recommendations):** Conjunto de padrões da comunidade PHP-FIG. Importantes:

- PSR-1/PSR-12: padrões de codificação (nomes de classes em StudlyCaps, constantes MAIÚSCULAS, métodos camelCase, indentação 4 espaços, etc). Seguir um padrão torna seu código consistente e integrável. (PSR-2 era antigo, substituído pelo PSR-12 atualizado).
- PSR-4: Autoload de classes via namespaces -> diretórios. Ex: Namespace `App\Models\User` corresponde a arquivo `src/Models/User.php`. Most modern frameworks adopt PSR-4.
- PSR-7: padrão de interface para requisição/resposta HTTP (usado em frameworks para interoperabilidade).

- **Segurança (SQL Injection, XSS):** Ao construir apps PHP, deve-se atentar a:

- *SQL Injection:* Sempre use consultas parametrizadas (prepared statements) ao inserir dados em SQL. Exemplo com PDO:

```
$stmt = $pdo->prepare("SELECT * FROM usuarios WHERE email = :email");
$stmt->execute(['email' => $emailDigitado]);
```

Isso evita que dados maliciosos fechem a query e executem SQL arbitrário.
- *XSS:* Ao exibir conteúdo vindo do usuário no HTML, sanitize (como usar `htmlspecialchars` para escapar `< > &`). Ou use ferramentas templating que fazem isso automaticamente. Validate também inputs no servidor (não confie só no front-end).
- *CSRF:* Use tokens de sessão em formulários para garantir que requisições vindas de outros sites não consigam forjar ações do usuário autenticado. Ex: gerar um `$_SESSION['token']` e inserir como `<input type="hidden" name="token">` nos forms, e verificar no POST recebido.

- *Senha:* Nunca armazene senha em texto puro. Sempre **hash** (com algoritmos como bcrypt). Use `password_hash("senha", PASSWORD_DEFAULT)` e verifique com `password_verify($senhaDigitada, $hashArmazenado)`. Isso automaticamente usa salts e um algoritmo forte configurado.

- **Exceptions Avançadas:** Além de Exception base class, existem outras (Error, etc). Em PHP 7+, muitos erros fatais são transformados em exceptions do tipo `Error` (por exemplo, erro de tipo, etc). Você pode capturar `Throwable` para pegar qualquer exceção ou erro. Custom exceptions: você pode estender Exception class para tipos específicos.

- **Traits:** Permitem adicionar métodos reutilizáveis a classes sem herança múltipla. Ex:

```
trait Logger {
  public function log($msg) {
    echo "[LOG]: $msg";
  }
```

```
}
class Usuario {
  use Logger;
  // ... agora Usuario tem o método log()
}
```

Traços podem conter métodos e propriedades. Se múltiplos traits introduzem método com mesmo nome, pode haver conflito e é possível resolver explicitamente qual usar. Traços ajudam a compartilhar código comum entre classes diferentes que não estão diretamente relacionadas.

- **Uso de Bibliotecas e Frameworks:** Em entrevistas, podem esperar que você pelo menos conheça os frameworks populares:

- **Laravel** (um dos mais usados atualmente) – um framework web completo seguindo MVC, com ORM (Eloquent), sistema de migrations, blade templates, etc.
- **Symfony** – outro poderoso framework, mais enterprise, que inclusive serve de base para Laravel em diversos componentes.
- Outros: CodeIgniter, Zend Framework/Laminas, CakePHP, Yii, etc.
- Microframeworks: Slim, Lumen (versão light do Laravel), etc., para APIs simples.

Saber frameworks indica que você entende estrutura MVC, roteamento, controllers, etc. Mas como o foco é linguagem, mencionar ao menos que existem e você daria ênfase no Laravel se perguntado, por exemplo.

- **Noções de HTTP e APIs:** Como PHP frequentemente roda em contexto web, entenda cabeçalhos HTTP, códigos de status (200, 404, 500, etc.), métodos (GET, POST, PUT, DELETE...). Em PHP, se criando API REST, você leria `$_SERVER['REQUEST_METHOD']` e manipularia I/O (talvez usando frameworks ou libs específicas). Saber enviar/ler JSON: usar `json_encode` para resposta, `json_decode` para request JSON (lido do `php://input`).

- **CLI PHP e Scripts:** PHP pode rodar em linha de comando para scripts utilitários ou cron jobs. Ex: `php meu_script.php`. Em CLI, `$_SERVER['argv']` dá argumentos. Pode usar libraries como Symfony Console ou implementações simples para argumentos.

- **Performance e Depuração:**

- OpCache: habilitar (no php.ini) caching de bytecode para melhorar desempenho em produção.
- Xdebug: extensão para debug, permite step-by-step debugging com IDEs, e geração de traces e profiles.

- Profiling: ferramentas como XHProf ou Tideways podem ser usadas para identificar gargalos.

- **Multibyte Strings:** Ao lidar com strings UTF-8 (acentos), use funções `mb_*` (mb_strlen, mb_substr, etc.) ou configure `default_charset='UTF-8'` e `mb_internal_encoding('UTF-8')`. Strings padrão do PHP podem truncar bytes se não cuidadoso, mb_* fns tratam caracteres multibyte corretamente.

- **Novidades do PHP 7/8:**

- PHP 7 introduziu tipagem escalar e de retorno opcionais. Ex:

```
function soma(int $a, int $b): int {
  return $a + $b;
}
```

Isso permite checagem de tipos (pode definir strict_types=1 no topo para tornar obrigatório).

- Operador coalesce `??` (retorna o primeiro operando não nulo) – útil para valores padrão: `$valor = $_GET['chave'] ?? 'default';`.
- Operador spaceship `<=>` para comparações tri-state (a<b => -1, a==b =>0, a>b =>1).
- Anonymous classes: `new class { ... }` on the fly classes.
- PHP 7.4: typed properties, arrow functions `fn($x) => $x*2;`, operador nullsafe `?->` (no PHP8?), etc.
- PHP 8: JIT compilation (melhora performance de CPU-bound tasks em alguns casos), Union types (function foo(int|float $x)), atributos (annotations nativas), nullsafe operator (`$obj?->prop` which returns null if $obj is null instead of error), match expression (similar to switch but expression-based), etc.

Exemplo de Código Avançado (combining concepts): Um exemplo para ilustrar alguns pontos:

```
<?php
namespace App\Utils;

class Calculadora {
  use \App\Traits\Loggable;  // imagine que Loggable trait fornece método
log()

  public static function somar(float ...$nums): float {
    $resultado = array_sum($nums);
    self::log("Somatório de ".json_encode($nums)." = $resultado");
    return $resultado;
  }
}

// Trait no namespace App\Traits
namespace App\Traits;
trait Loggable {
  protected static function log($msg) {
    // escrevendo log simples em arquivo
    file_put_contents(__DIR__."/app.log", date('c')." - $msg\n",
FILE_APPEND);
  }
}
?>
```

(Esse código estaria distribuído em arquivos apropriados, autoload via Composer, etc.)

Aqui vemos: - Uso de namespaces para organizar classes e traits. - Trait Loggable com método de logging protegido. - Classe Calculadora usando trait Loggable. - Método static somar usando tipagem estrita (float ...$nums significa qualquer quantidade de floats) e retornando float. Usa array_sum para

somar e chama self::log (do trait) para registrar. - file_put_contents com FILE_APPEND para adicionar logs com timestamp (date('c') formato ISO8601 datetime).

Isso é um pedaço de lógica, que poderíamos chamar em index.php:

```
use App\Utils\Calculadora;
require 'vendor/autoload.php';  // supondo autoload configurado
echo Calculadora::somar(10.5, 2, 3.5);
```

Esse mostraria 16 (e registraria log). Demonstra estáticos, variadic parameters, JSON encoding de array, etc.

Exercícios

1. **PDO com Prepared Statements:** Escreva um código PHP que insere um novo registro em uma tabela "users" (colunas: nome, email) usando PDO e prepared statements. Suponha que já há uma conexão `$pdo` estabelecida. Mostre como usar named parameters (:nome, :email) e tratar possíveis exceções.
2. **Sugestão de Solução:**

   ```
   <?php
     try {
       $pdo = new PDO("mysql:host=localhost;dbname=teste;charset=utf8",
   "usuario", "senha");
       $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
       $sql = "INSERT INTO users (nome, email) VALUES (:nome, :email)";
       $stmt = $pdo->prepare($sql);
       $nome = "João da Silva";
       $email = "joao.silva@example.com";
       $stmt->bindParam(':nome', $nome);
       $stmt->bindParam(':email', $email);
       $stmt->execute();
       echo "Usuário inserido com ID: " . $pdo->lastInsertId();
     } catch (PDOException $e) {
       echo "Erro ao inserir usuário: " . $e->getMessage();
     }
   ?>
   ```

   Explicação: Iniciamos a conexão PDO (num cenário real, isso estaria separado). Preparamos o INSERT com placeholders nomeados. Usamos `$stmt->bindParam` para vincular as variáveis $nome e $email aos placeholders (bindParam liga por referência, aqui ok; poderíamos usar bindValue também). Executamos. Se sucesso, usamos `$pdo->lastInsertId()` para pegar o ID auto-increment gerado e exibimos. Em caso de erro, catch captura e mostra mensagem. O uso de prepared statements (`prepare` + `execute`) protege contra SQL injection automaticamente para esses campos e é boa prática. Esse exercício demonstra interação básica com BD em modo seguro.
3. **Upload de Arquivo:** Escreva um script PHP que processe o upload de um arquivo enviado via formulário (campo `file` com name "arquivo"). O script deve verificar se o upload ocorreu sem erros, mover o arquivo para um diretório "uploads/" com um nome único (por exemplo,

concatenando timestamp ou usando `uniqid`), e exibir uma mensagem de sucesso com o caminho do arquivo salvo, ou mensagem de erro em caso de falha.

4. **Sugestão de Solução:**

```php
<?php
  if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (isset($_FILES['arquivo']) && $_FILES['arquivo']['error'] ===
UPLOAD_ERR_OK) {
      $nomeTemp = $_FILES['arquivo']['tmp_name'];
      $nomeOriginal = basename($_FILES['arquivo']['name']);
      $extensao = pathinfo($nomeOriginal, PATHINFO_EXTENSION);
      $novoNome = uniqid("upload_").".".$extensao;
      $destino = __DIR__ . "/uploads/" . $novoNome;
      if (move_uploaded_file($nomeTemp, $destino)) {
        echo "Upload concluído com sucesso! Arquivo disponível em:
uploads/$novoNome";
      } else {
        echo "Erro: Falha ao mover o arquivo enviado.";
      }
    } else {
      echo "Nenhum arquivo enviado ou erro no upload (Código: " .
($_FILES['arquivo']['error'] ?? 'N/A') . ").";
    }
  }
?>
```

Explicação: Primeiro verificamos se o método da requisição é POST (para garantir que veio de um form submission). Depois verificamos `$_FILES['arquivo']` e se `error == UPLOAD_ERR_OK` (valor 0) indicativo de sucesso no upload. Cada arquivo enviado aparece nesse array superglobal com campos `name`, `tmp_name`, `size`, `error`, `type`. Pegamos o caminho temporário (`tmp_name`) onde o PHP armazenou. Pegamos o nome original do arquivo com `basename` (segurança: basename remove diretórios no nome). Extraímos a extensão do arquivo. Geramos um novo nome único usando `uniqid` (prefixado com "upload_") e acrescentamos a extensão original. Definimos $destino como pasta uploads/ + novoNome (usando **DIR** para pegar diretório atual do script, por segurança). Então `move_uploaded_file` para mover do tmp para o destino final. Esse func retorna true se sucesso. Se moveu, exibimos mensagem com caminho relativo do arquivo. Se falhou ou se não tinha arquivo/encontrou erro, tratamos e exibimos mensagem com código do erro. Observação: é preciso garantir que o diretório uploads/ exista e tenha permissões de escrita, e idealmente validar o tipo do arquivo para segurança (não deixar subir .php por exemplo). Mas aqui focamos no básico do upload handling. Esse exercício passa por $_FILES usage, uniqid, pathinfo – coisas típicas ao lidar com arquivos no PHP.

5. **Aplicando Conceitos OOP Avançados:** Suponha uma interface `Formattable` com um método `format()` definido. Crie uma classe `Produto` que implementa essa interface. A classe deve ter propriedades privadas `nome` e `preco`. Forneça métodos getters/setters para nome e preco (com validações simples, ex: preço não negativo). Implemente o método `format()` para retornar uma string formatada do produto, por exemplo: `"Produto: Nome - R$ 123.45"`. Demonstre a criação de um objeto `Produto`, atribuição de valores via setters e saída do `format()`.

6. **Sugestão de Solução:**

```php
<?php
  interface Formattable {
    public function format(): string;
  }

  class Produto implements Formattable {
    private $nome;
    private $preco;

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

    public function getNome() {
      return $this->nome;
    }
    public function setNome($nome) {
      $nome = trim($nome);
      if ($nome === "") {
        throw new InvalidArgumentException("Nome não pode ser vazio");
      }
      $this->nome = $nome;
    }
    public function getPreco() {
      return $this->preco;
    }
    public function setPreco($preco) {
      if (!is_numeric($preco) || $preco < 0) {
        throw new InvalidArgumentException("Preço inválido");
      }
      $this->preco = (float)$preco;
    }
    public function format(): string {
      // number_format para formatar com duas casas decimais e vírgula
      $precoFmt = "R$ " . number_format($this->preco, 2, ',', '.');
      return "Produto: {$this->nome} - {$precoFmt}";
    }
  }

  // Demonstração
  try {
    $prod = new Produto("Cafeteira", 299.99);
    echo $prod->format();
  } catch(Exception $e) {
    echo "Erro ao criar produto: " . $e->getMessage();
  }
?>
```

Explicação: Definimos a interface `Formattable` com método `format()`. A classe `Produto` implementa essa interface, então deve ter public function format(): string. Propriedades nome e preco são privadas. Construtor __construct chama os setters para centralizar validação. O setNome remove espaços e checa se não fica vazio, caso contrário lança InvalidArgumentException (uma exceção padrão do SPL). setPreco checa se é numérico e >=0, senão lança exceção. Armazena preco como float. O format() utiliza number_format para formatar o preço com 2 casas decimais e vírgula decimal/ponto milhar padrão Brasil. Retorna uma string com nome e preço formatado. Depois, no uso, criamos um Produto (dentro de try/catch para pegar possíveis exceções das validações). Se sucesso, exibimos o format dele. Para "Cafeteira" e 299.99, a saída seria: `Produto: Cafeteira - R$ 299,99`. Esse exercício cobre: interface, implementação, encapsulamento com private + getters/setters, validação com exceção, uso de try/catch, e formatação numérica. Mostra noções avançadas de OOP e robustez de código.

Com esta revisão completa, você deve ter reforçado os principais conceitos de HTML, CSS, JavaScript e PHP, desde fundamentos até tópicos mais avançados. Cada seção abordou teorias essenciais, exemplos práticos comentados e exercícios que refletem casos de uso reais. Ao dominar esses conteúdos e práticas, você estará mais preparado para construir projetos web modernos e enfrentar eventuais perguntas técnicas em entrevistas, especialmente no que tange ao PHP, que recebeu atenção especial. Bons estudos e boa sorte em sua entrevista!

---

1 8 9 10 11 12 13 Entrevistas PHP | 56 Perguntas e Respostas para 2024
https://www.turing.com/pt/interview-questions/php

2 6 7 Melhores Práticas de HTML para Criação de Sites Manejáveis e Escaláveis
https://kinsta.com/pt/blog/melhores-praticas-html/

3 4 5 HTML: Boas práticas em acessibilidade - Aprendendo desenvolvimento web | MDN
https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/Accessibility/HTML