

Contents

Guia Completo de JavaScript, TypeScript e PHP	1
1. Introdução	2
Sobre este guia	2
Versões das linguagens	2
Documentação oficial	2
2. Operadores	2
2.1. Operadores Aritméticos	3
2.2. Operadores de Atribuição	4
2.3. Operadores de Comparação	6
2.4. Operadores Lógicos	8
2.5. Operadores de String	9
2.6. Operadores de Incremento/Decremento	10
2.7. Outros Operadores	11
2.8. Diferença de Tipagem: JavaScript vs. TypeScript	17
3. Estruturas de Controle	19
3.1. Estruturas Condicionais	19
3.2. Estruturas de Repetição	24
3.3. Controle de Fluxo	28
4. Programação Orientada a Objetos (POO)	30
Conceitos Fundamentais da POO (Classes, Objetos, Atributos, Métodos)	30
Encapsulamento	33
Herança	37
Polimorfismo	40
Interfaces e Classes Abstratas (onde aplicável)	42
5. Referencias Usadas	47
Orientação a Objetos por Prototipação (JavaScript)	47
Recursos Adicionais:	51
6. Introdução a Laravel, React Native e SQL	51
6.2. Laravel	51
6.3. React Native	53
6.1. SQL Portátil: Comandos Essenciais	57

Guia Completo de JavaScript, TypeScript e PHP

Este guia tem como objetivo fornecer um material de consulta completo e organizado sobre operadores, estruturas de controle e programação orientada a objetos nas linguagens JavaScript,

TypeScript e PHP. O foco é apresentar os conceitos de forma integrada, destacando as semelhanças e diferenças de sintaxe e comportamento entre as linguagens, sempre com base nas versões mais recentes e estáveis.

1. Introdução

Sobre este guia

Este guia foi elaborado para desenvolvedores que trabalham com MySQL, Laravel e React Native, e buscam um material de consulta rápido e eficiente para revisar os fundamentos de JavaScript, TypeScript e PHP. A abordagem é focada na integração e comparação entre as linguagens, facilitando a compreensão das semelhanças e diferenças sintáticas e conceituais. O conteúdo é organizado por capítulos, com links diretos para a documentação oficial, garantindo a precisão e a atualização das informações.

Versões das linguagens

Para garantir a relevância e a precisão das informações, este guia se baseia nas versões mais recentes e estáveis das linguagens no momento de sua criação:

- JavaScript (Node.js LTS): v22 (LTS - “Jod”)
- TypeScript: 5.9.2
- PHP: 8.3 (versão estável atual), com menções à 8.4 (futura estável) quando relevante.

Documentação oficial

Para aprofundamento e consulta detalhada, recomenda-se sempre a documentação oficial das respectivas linguagens:

- JavaScript: MDN Web Docs - JavaScript
- TypeScript: TypeScript Documentation
- PHP: Manual do PHP

2. Operadores

Operadores são símbolos que instruem o interpretador a realizar alguma operação matemática ou lógica. Eles são fundamentais em qualquer linguagem de programação para manipular valores e controlar o fluxo de execução. Embora JavaScript, TypeScript e PHP compartilhem muitos operadores em comum devido às suas raízes e influências, existem diferenças sutis e específicas de cada linguagem que serão abordadas nesta seção.

2.1. Operadores Aritméticos

Utilizados para realizar operações matemáticas básicas.

Operador	Descrição	Exemplo (JS/TS)	Exemplo (PHP)
+	Adição	5 + 3	5 + 3
-	Subtração	5 - 3	5 - 3
*	Multiplicação	5 * 3	5 * 3
/	Divisão	5 / 3	5 / 3
%	Módulo (Resto)	5 % 3	5 % 3
**	Exponenciação	5 ** 3	5 ** 3

JavaScript/TypeScript:

JavaScript:

```
1 let a = 10;
2 let b = 3;
3 console.log(a + b); // 13
4 console.log(a - b); // 7
5 console.log(a * b); // 30
6 console.log(a / b); // 3.333...
7 console.log(a % b); // 1
8 console.log(a ** b); // 1000 (10 elevado à potência de 3)
```

TypeScript:

```
1 let a: number = 10;
2 let b: number = 3;
3 console.log(a + b); // 13
4 console.log(a - b); // 7
5 console.log(a * b); // 30
6 console.log(a / b); // 3.333...
7 console.log(a % b); // 1
8 console.log(a ** b); // 1000 (10 elevado à potência de 3)
```

PHP:

```
1 $a = 10;
2 $b = 3;
3 echo $a + $b; // 13
4 echo $a - $b; // 7
5 echo $a * $b; // 30
6 echo $a / $b; // 3.333...
7 echo $a % $b; // 1
```

```
8 echo $a ** $b; // 1000 (10 elevado à potência de 3)
```

Observações:

- Todos os operadores aritméticos funcionam de forma similar nas três linguagens.
- Em JavaScript/TypeScript, a divisão por zero resulta em `Infinity` ou `NaN` (Not-a-Number), dependendo do operando. Em PHP, a divisão por zero gera um aviso (Warning) e o resultado é `INF` (Infinity) ou `NAN`.

Documentação Oficial:

- JavaScript: Operadores Aritméticos
- PHP: Operadores Aritméticos

2.2. Operadores de Atribuição

Utilizados para atribuir valores a variáveis.

Operador	Exemplo (JS/TS)	Equivalente (JS/TS)	Exemplo (PHP)	Equivalente (PHP)
<code>=</code>	<code>x = 5</code>	<code>x = 5</code>	<code>\$x = 5</code>	<code>\$x = 5</code>
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>	<code>\$x += 5</code>	<code>\$x = \$x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>	<code>\$x -= 5</code>	<code>\$x = \$x - 5</code>
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>	<code>\$x *= 5</code>	<code>\$x = \$x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>	<code>\$x /= 5</code>	<code>\$x = \$x / 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>	<code>\$x %= 5</code>	<code>\$x = \$x % 5</code>
<code>**=</code>	<code>x **= 5</code>	<code>x = x ** 5</code>	<code>\$x **= 5</code>	<code>\$x = \$x ** 5</code>

JavaScript/TypeScript:

JavaScript:

```
1 let x = 10;  
2 x += 5; // x agora é 15  
3 console.log(x);  
4 x -= 3; // x agora é 12  
5 console.log(x);  
6 x *= 2; // x agora é 24  
7 console.log(x);
```

```
8 x /= 4; // x agora é 6
9 console.log(x);
10 x %= 5; // x agora é 1
11 console.log(x);
12 x **= 3; // x agora é 1 (1 elevado à potência de 3)
13 console.log(x);
```

TypeScript:

```
1 let x: number = 10;
2 x += 5; // x agora é 15
3 console.log(x);
4 x -= 3; // x agora é 12
5 console.log(x);
6 x *= 2; // x agora é 24
7 console.log(x);
8 x /= 4; // x agora é 6
9 console.log(x);
10 x %= 5; // x agora é 1
11 console.log(x);
12 x **= 3; // x agora é 1 (1 elevado à potência de 3)
13 console.log(x);
```

PHP:

```
1 $x = 10;
2 $x += 5; // $x agora é 15
3 echo $x . "\n";
4 $x -= 3; // $x agora é 12
5 echo $x . "\n";
6 $x *= 2; // $x agora é 24
7 echo $x . "\n";
8 $x /= 4; // $x agora é 6
9 echo $x . "\n";
10 $x %= 5; // $x agora é 1
11 echo $x . "\n";
12 $x **= 3; // $x agora é 1 (1 elevado à potência de 3)
13 echo $x . "\n";
```

Observações:

- Os operadores de atribuição compostos (+=, -=, etc.) funcionam de maneira idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: Operadores de Atribuição

- PHP: Operadores de Atribuição

2.3. Operadores de Comparação

Utilizados para comparar dois valores e retornar um valor booleano (`true` ou `false`).

Operador	Descrição	Exemplo (JS/TS)	Exemplo (PHP)
<code>=</code>	Igual a (com conversão de tipo)	<code>5 = '5'</code>	<code>5 = '5'</code>
<code>===</code>	Estritamente igual a (sem conversão)	<code>5 === '5'</code>	<code>5 === '5'</code>
<code>≠</code>	Diferente de (com conversão de tipo)	<code>5 ≠ '5'</code>	<code>5 ≠ '5'</code>
<code>≡</code>	Estritamente diferente de (sem conversão)	<code>5 ≡ '5'</code>	<code>5 ≡ '5'</code>
<code>></code>	Maior que	<code>5 > 3</code>	<code>5 > 3</code>
<code><</code>	Menor que	<code>5 < 3</code>	<code>5 < 3</code>
<code>≥</code>	Maior ou igual a	<code>5 ≥ 3</code>	<code>5 ≥ 3</code>
<code>≤</code>	Menor ou igual a	<code>5 ≤ 3</code>	<code>5 ≤ 3</code>
<code>≤ ></code>	Spaceship (PHP 7+)	N/A	<code>1 ≤ > 2</code>

JavaScript/TypeScript:

JavaScript:

```

1 console.log(5 == '5'); // true (coerção de tipo)
2 console.log(5 === '5'); // false (tipos diferentes)
3 console.log(5 != '5'); // false
4 console.log(5 !== '5'); // true
5 console.log(10 > 5); // true
6 console.log(10 < 5); // false
7 console.log(10 >= 10); // true
8 console.log(10 <= 5); // false

```

TypeScript:

```

1 // TypeScript permite a comparação, mas o linter/compilador pode alertar sobre comparações
  de tipos diferentes
2 console.log(5 == '5'); // true (coerção de tipo)
3 console.log(5 === '5'); // false (tipos diferentes)

```

```

4 console.log(5 !== '5'); // false
5 console.log(5 !== '5'); // true
6 console.log(10 > 5); // true
7 console.log(10 < 5); // false
8 console.log(10 >= 10); // true
9 console.log(10 <= 5); // false
10
11 // Exemplo com tipagem explícita para clareza
12 let num1: number = 5;
13 let str1: string = '5';
14
15 // console.log(num1 == str1); // Erro em tempo de compilação se strictEqualityChecks
    estiver ativado
16 console.log(num1 === Number(str1)); // true (conversão explícita)

```

PHP:

```

1 var_dump(5 == '5'); // bool(true)
2 var_dump(5 === '5'); // bool(false)
3 var_dump(5 != '5'); // bool(false)
4 var_dump(5 !== '5'); // bool(true)
5 var_dump(10 > 5); // bool(true)
6 var_dump(10 < 5); // bool(false)
7 var_dump(10 >= 10); // bool(true)
8 var_dump(10 <= 5); // bool(false)
9
10 // Operador Spaceship (PHP 7+)
11 var_dump(1 <=> 1); // int(0) (igual)
12 var_dump(1 <=> 2); // int(-1) (esquerda é menor)
13 var_dump(2 <=> 1); // int(1) (esquerda é maior)

```

Observações:

- **Coerção de Tipo:** JavaScript e PHP realizam coerção de tipo com `=` e `≠`. Isso significa que eles tentam converter os operandos para um tipo comum antes da comparação. TypeScript, por ser um superconjunto de JavaScript, herda esse comportamento, mas o uso de tipos estáticos em TypeScript geralmente ajuda a evitar comparações que dependem de coerção implícita.
- **Comparação Estrita (`===`, `!==`):** É altamente recomendado usar os operadores de comparação estrita (`===` e `!==`) em JavaScript, TypeScript e PHP, pois eles comparam tanto o valor quanto o tipo, evitando comportamentos inesperados devido à coerção de tipo.
- **Operador Spaceship (`<=>`):** Exclusivo do PHP (a partir da versão 7). Ele retorna `0` se os operandos forem iguais, `-1` se o operando da esquerda for menor, e `1` se o operando da esquerda for maior. É útil para funções de comparação (callbacks) em ordenação.

Documentação Oficial:

- JavaScript: Operadores de Comparação
- PHP: Operadores de Comparação

2.4. Operadores Lógicos

Utilizados para combinar ou negar expressões booleanas.

Operador	Descrição	Exemplo (JS/TS)	Exemplo (PHP)
<code>&&</code>	AND lógico	<code>a && b</code>	<code>\$a && \$b</code>
			OR lógico

JavaScript/TypeScript:

```
1 let idade = 20;
2 let temHabilitacao = true;
3 console.log(idade >= 18 && temHabilitacao); // true (ambas as condições são verdadeiras)
4 console.log(idade < 18 || temHabilitacao); // true (pelo menos uma condição é verdadeira)
5 console.log(!temHabilitacao); // false (negação de true)
```

TypeScript:

```
1 let idade: number = 20;
2 let temHabilitacao: boolean = true;
3 console.log(idade >= 18 && temHabilitacao); // true (ambas as condições são verdadeiras)
4 console.log(idade < 18 || temHabilitacao); // true (pelo menos uma condição é verdadeira)
5 console.log(!temHabilitacao); // false (negação de true)
```

PHP:

```
1 $idade = 20;
2 $temHabilitacao = true;
3 var_dump($idade >= 18 && $temHabilitacao); // bool(true)
4 var_dump($idade < 18 || $temHabilitacao); // bool(true)
5 var_dump(!$temHabilitacao); // bool(false)
```

Observações:

- Os operadores lógicos (`&&`, `||`, `!`) funcionam de forma idêntica nas três linguagens, incluindo o comportamento de curto-circuito (short-circuit evaluation), onde a segunda expressão só é avaliada se a primeira não for suficiente para determinar o resultado final.

Documentação Oficial:

- JavaScript: Operadores Lógicos
- PHP: Operadores Lógicos

2.5. Operadores de String

Utilizados para manipular strings.

Operador	Descrição	Exemplo (JS/TS)	Exemplo (PHP)
+	Concatenação (JS/TS)	'Olá' + ' Mundo'	N/A
.	Concatenação (PHP)	N/A	'Olá' . ' Mundo'
+=	Atribuição com Concatenação (JS/TS)	str += '!'	N/A
.=	Atribuição com Concatenação (PHP)	N/A	\$str .= '!'

JavaScript/TypeScript:

```
1 let saudacao = 'Olá';
2 let nome = 'Mundo';
3 let mensagem = saudacao + ' ' + nome; // Olá Mundo
4 console.log(mensagem);
5 saudacao += '!'; // Olá!
6 console.log(saudacao);
```

TypeScript:

```
1 let saudacao: string = 'Olá';
2 let nome: string = 'Mundo';
3 let mensagem: string = saudacao + ' ' + nome; // Olá Mundo
4 console.log(mensagem);
5 saudacao += '!'; // Olá!
6 console.log(saudacao);
```

PHP:

```
1 $saudacao = 'Olá';
2 $nome = 'Mundo';
3 $mensagem = $saudacao . ' ' . $nome; // Olá Mundo
4 echo $mensagem . "\n";
5 $saudacao .= '!'; // Olá!
6 echo $saudacao . "\n";
```

Observações:

- **JavaScript/TypeScript:** Utilizam o operador `+` para concatenação de strings. Se um dos operandos for uma string e o outro não, o outro operando será convertido para string antes da concatenação.
- **PHP:** Utiliza o operador `.` para concatenação de strings. O operador `+` em PHP é estritamente para operações aritméticas.

Documentação Oficial:

- JavaScript: Operador de Concatenação
- PHP: Operadores de String

2.6. Operadores de Incremento/Decremento

Utilizados para aumentar ou diminuir o valor de uma variável em uma unidade.

Operador	Descrição	Exemplo (JS/TS)	Exemplo (PHP)
<code>++</code>	Incremento	<code>x++</code> ou <code>++x</code>	<code>\$x++</code> ou <code>++\$x</code>
<code>--</code>	Decremento	<code>x--</code> ou <code>--x</code>	<code>\$x--</code> ou <code>--\$x</code>

JavaScript/TypeScript:

```

1 let contador = 0;
2 console.log(contador++); // 0 (pós-incremento: usa o valor atual, depois incrementa)
3 console.log(contador);   // 1
4 console.log(++contador); // 2 (pré-incremento: incrementa, depois usa o novo valor)
5 console.log(contador);   // 2
6
7 let valor = 10;
8 console.log(valor--);    // 10 (pós-decremento: usa o valor atual, depois decrementa)
9 console.log(valor);      // 9
10 console.log(--valor);    // 8 (pré-decremento: decrementa, depois usa o novo valor)
11 console.log(valor);     // 8

```

TypeScript:

```

1 let contador: number = 0;
2 console.log(contador++); // 0 (pós-incremento: usa o valor atual, depois incrementa)
3 console.log(contador);   // 1
4 console.log(++contador); // 2 (pré-incremento: incrementa, depois usa o novo valor)
5 console.log(contador);   // 2
6

```

```

7 let valor: number = 10;
8 console.log(valor--);    // 10 (pós-decremento: usa o valor atual, depois decrementa)
9 console.log(valor);      // 9
10 console.log(--valor);    // 8 (pré-decremento: decrementa, depois usa o novo valor)
11 console.log(valor);     // 8

```

PHP:

```

1 $contador = 0;
2 echo $contador++ . "\n"; // 0 (pós-incremento)
3 echo $contador . "\n";   // 1
4 echo ++$contador . "\n"; // 2 (pré-incremento)
5 echo $contador . "\n";   // 2
6
7 $valor = 10;
8 echo $valor-- . "\n";    // 10 (pós-decremento)
9 echo $valor . "\n";      // 9
10 echo --$valor . "\n";   // 8 (pré-decremento)
11 echo $valor . "\n";     // 8

```

Observações:

- O comportamento de pré-incremento/decremento e pós-incremento/decremento é idêntico nas três linguagens.

Documentação Oficial:

- JavaScript: Operadores de Incremento e Decremento
- PHP: Operadores de Incremento/Decremento

2.7. Outros Operadores

Operador Ternário (Condicional) Um atalho para a instrução `if ... else`.

Operador	Descrição	Sintaxe	Exemplo (JS/TS)	Exemplo (PHP)
<code>? :</code>	Ternário	<code>condicao ?</code> <code>expressao1 :</code> <code>expressao2</code>	<code>idade ≥ 18 ? 'Maior' : 'Menor'</code>	<code>\$idade ≥ 18 ? 'Maior' : 'Menor'</code>

JavaScript/TypeScript:

```
1 let idade = 20;
2 let status = (idade >= 18) ? 'Maior de idade' : 'Menor de idade';
3 console.log(status); // Maior de idade
```

TypeScript:

```
1 let idade: number = 20;
2 let status: string = (idade >= 18) ? 'Maior de idade' : 'Menor de idade';
3 console.log(status); // Maior de idade
```

PHP:

```
1 $idade = 20;
2 $status = ($idade >= 18) ? 'Maior de idade' : 'Menor de idade';
3 echo $status . "\n"; // Maior de idade
```

Observações:

- O operador ternário funciona de forma idêntica nas três linguagens, oferecendo uma forma concisa de expressar condicionais simples.

Documentação Oficial:

- JavaScript: Operador Condicional (Ternário)
- PHP: Operador Ternário

Operador typeof (JavaScript/TypeScript) Retorna uma string indicando o tipo do operando.

JavaScript/TypeScript:

```
1 console.log(typeof 42);           // "number"
2 console.log(typeof "hello");      // "string"
3 console.log(typeof true);         // "boolean"
4 console.log(typeof undefined);    // "undefined"
5 console.log(typeof null);         // "object" (um erro histórico do JavaScript)
6 console.log(typeof {});           // "object"
7 console.log(typeof []);           // "object"
8 console.log(typeof function(){}); // "function"
```

TypeScript:

```
1 console.log(typeof 42);           // "number"
2 console.log(typeof "hello");      // "string"
3 console.log(typeof true);         // "boolean"
4 console.log(typeof undefined);    // "undefined"
```

```

5 console.log(typeof null);          // "object" (um erro histórico do JavaScript)
6 console.log(typeof {});           // "object"
7 console.log(typeof []);           // "object"
8 console.log(typeof function(){}); // "function"
9
10 // Em TypeScript, a tipagem estática permite que você saiba o tipo de uma variável
11 // antes mesmo de usar `typeof` em tempo de execução.
12 let num: number = 42;
13 let str: string = "hello";
14 let bool: boolean = true;
15 let undef: undefined = undefined;
16 let nulo: null = null;
17 let obj: object = {};
18 let arr: any[] = [];
19 let func: Function = function(){};
20
21 // O uso de `typeof` em TypeScript é mais para inspeção em tempo de execução,
22 // já que o compilador já conhece os tipos.
23 console.log(`Tipo de num: ${typeof num}`);
24 console.log(`Tipo de str: ${typeof str}`);
25 console.log(`Tipo de bool: ${typeof bool}`);
26 console.log(`Tipo de undef: ${typeof undef}`);
27 console.log(`Tipo de nulo: ${typeof nulo}`);
28 console.log(`Tipo de obj: ${typeof obj}`);
29 console.log(`Tipo de arr: ${typeof arr}`);
30 console.log(`Tipo de func: ${typeof func}`);

```

Observações:

- O operador `typeof` é útil para verificar tipos primitivos. Para objetos, ele retorna "object" (exceto para funções). Para `null`, ele retorna "object", o que é uma peculiaridade histórica do JavaScript.

Documentação Oficial:

- JavaScript: Operador `typeof`

Operador `instanceof` (JavaScript/TypeScript) Verifica se um objeto é uma instância de uma classe ou de um tipo de objeto específico.

JavaScript/TypeScript:

```

1 class Animal {
2   constructor(nome) {
3     this.nome = nome;

```

```

4   }
5 }
6
7 class Cachorro extends Animal {
8   constructor(nome, raca) {
9     super(nome);
10    this.raca = raca;
11  }
12 }
13
14 let meuCachorro = new Cachorro("Rex", "Labrador");
15 let meuAnimal = new Animal("Leão");
16
17 console.log(meuCachorro instanceof Cachorro); // true
18 console.log(meuCachorro instanceof Animal);   // true
19 console.log(meuAnimal instanceof Cachorro);   // false
20 console.log([] instanceof Array);              // true
21 console.log({} instanceof Object);             // true

```

TypeScript:

```

1 class AnimalTS {
2   constructor(public nome: string) {}
3 }
4
5 class CachorroTS extends AnimalTS {
6   constructor(nome: string, public raca: string) {
7     super(nome);
8   }
9 }
10
11 let meuCachorroTS: CachorroTS = new CachorroTS("Rex", "Labrador");
12 let meuAnimalTS: AnimalTS = new AnimalTS("Leão");
13
14 console.log(meuCachorroTS instanceof CachorroTS); // true
15 console.log(meuCachorroTS instanceof AnimalTS);   // true
16 console.log(meuAnimalTS instanceof CachorroTS);   // false
17 console.log([] instanceof Array);                 // true
18 console.log({} instanceof Object);                 // true
19
20 // Em TypeScript, você pode usar a tipagem para garantir que um objeto
21 // é de um determinado tipo, o que é verificado em tempo de compilação.
22 function processarAnimal(animal: AnimalTS) {
23   if (animal instanceof CachorroTS) {
24     console.log(`É um cachorro da raça: ${animal.raca}`);
25   } else {
26     console.log(`É um animal genérico: ${animal.nome}`);

```

```
27     }  
28 }  
29  
30 processarAnimal(meuCachorroTS);  
31 processarAnimal(meuAnimalTS);
```

Observações:

- O operador `instanceof` verifica a cadeia de protótipos do objeto. Ele retorna `true` se o protótipo do construtor estiver presente na cadeia de protótipos do objeto.

Documentação Oficial:

- JavaScript: Operador `instanceof`

Funções de Verificação de Tipo em PHP PHP não possui operadores `typeof` ou `instanceof` no mesmo sentido que JavaScript/TypeScript. Em vez disso, ele oferece uma série de funções para verificar o tipo de uma variável e o operador `instanceof` para verificar a instância de uma classe.

Funções de Verificação de Tipo:

- `gettype()`: Retorna o tipo da variável como uma string.
- `is_array()`: Verifica se a variável é um array.
- `is_bool()`: Verifica se a variável é um booleano.
- `is_callable()`: Verifica se o conteúdo da variável pode ser chamado como uma função.
- `is_float()`: Verifica se a variável é um float (número de ponto flutuante).
- `is_int()`: Verifica se a variável é um inteiro.
- `is_null()`: Verifica se a variável é `NULL`.
- `is_numeric()`: Verifica se a variável é um número ou uma string numérica.
- `is_object()`: Verifica se a variável é um objeto.
- `is_resource()`: Verifica se a variável é um recurso.
- `is_string()`: Verifica se a variável é uma string.

PHP:

```
1 $numero = 123;
2 $texto = "Olá";
3 $booleano = true;
4 $array = [1, 2, 3];
5 $objeto = new stdClass();
6
7 echo gettype($numero) . "\n"; // integer
8 echo gettype($texto) . "\n";  // string
9 echo gettype($booleano) . "\n"; // boolean
10 echo gettype($array) . "\n";  // array
11 echo gettype($objeto) . "\n"; // object
12
13 var_dump(is_int($numero));    // bool(true)
14 var_dump(is_string($texto));  // bool(true)
15 var_dump(is_array($array));   // bool(true)
16 var_dump(is_object($objeto)); // bool(true)
```

Operador instanceof** (PHP):**

```
1 class Animal {}
2 class Cachorro extends Animal {}
3
4 $meuCachorro = new Cachorro();
5 $meuAnimal = new Animal();
6
7 var_dump($meuCachorro instanceof Cachorro); // bool(true)
8 var_dump($meuCachorro instanceof Animal);  // bool(true)
9 var_dump($meuAnimal instanceof Cachorro);   // bool(false)
```

Observações:

- Em PHP, `gettype()` retorna o tipo da variável como uma string, enquanto as funções `is_...()` retornam um booleano. O operador `instanceof` em PHP funciona de forma similar ao JavaScript/TypeScript para verificar a instância de uma classe.

Documentação Oficial:

- PHP: Funções de Tipo
- PHP: Operador instanceof

2.8. Diferença de Tipagem: JavaScript vs. TypeScript

Uma das distinções mais fundamentais entre JavaScript e TypeScript reside em seus sistemas de tipagem. Compreender essa diferença é crucial para escrever código robusto e escalável, especialmente em projetos maiores.

JavaScript: Tipagem Dinâmica e Fraca JavaScript é uma linguagem de **tipagem dinâmica e fraca**.

- **Tipagem Dinâmica (Dynamic Typing):** O tipo de uma variável é determinado em tempo de execução, não em tempo de compilação. Isso significa que você pode reatribuir uma variável com um valor de um tipo diferente sem gerar um erro.
- **Tipagem Fraca (Weak Typing / Loose Typing):** JavaScript permite conversões de tipo implícitas (coerção de tipo) em muitas operações. Isso pode levar a comportamentos inesperados se você não estiver ciente das regras de coerção.

Vantagens da Tipagem Dinâmica/Fraca (JavaScript):

- **Flexibilidade:** Permite prototipagem rápida e código mais conciso.
- **Menos verbosidade:** Não é necessário declarar tipos explicitamente.

Desvantagens da Tipagem Dinâmica/Fraca (JavaScript):

- **Erros em tempo de execução:** Muitos erros relacionados a tipos só são descobertos quando o código é executado, o que pode dificultar a depuração.
- **Manutenção:** Em projetos grandes, a falta de informações de tipo pode tornar o código mais difícil de entender, manter e refatorar.
- **Previsibilidade:** O comportamento de coerção de tipo pode ser confuso e levar a bugs sutis.

TypeScript: Tipagem Estática Opcional TypeScript é um superconjunto de JavaScript que adiciona **tipagem estática opcional**.

- **Tipagem Estática (Static Typing):** Os tipos das variáveis são verificados em tempo de compilação (antes da execução do código). Isso significa que o compilador TypeScript pode identificar muitos erros relacionados a tipos antes mesmo de o código ser executado.

- **Tipagem Opcional:** Você não é obrigado a usar tipos em todas as variáveis. TypeScript pode inferir tipos automaticamente (inferência de tipo) se você não os declarar explicitamente. Isso permite uma transição gradual de JavaScript para TypeScript.
- **Tipagem Forte (Strong Typing):** Embora TypeScript compile para JavaScript (que é fracamente tipado), ele impõe regras de tipo mais rigorosas em tempo de compilação, reduzindo a chance de coerções de tipo inesperadas.

Vantagens da Tipagem Estática (TypeScript):

- **Detecção precoce de erros:** Muitos bugs são pegos durante o desenvolvimento, antes que o código chegue à produção.
- **Melhor legibilidade e manutenção:** O código com tipos explícitos é mais fácil de entender, especialmente em equipes e projetos grandes.
- **Refatoração segura:** O compilador ajuda a garantir que as refatorações não quebrem o código.
- **Melhor autocompletar e ferramentas de desenvolvimento:** IDEs e editores de código podem fornecer sugestões mais precisas e navegação de código aprimorada.
- **Documentação implícita:** Os tipos servem como uma forma de documentação para o código.

Desvantagens da Tipagem Estática (TypeScript):

- **Curva de aprendizado:** Requer um tempo para aprender a sintaxe e os conceitos de tipagem.
- **Mais verbosidade:** Pode tornar o código um pouco mais longo devido às declarações de tipo.
- **Etapas de compilação:** Adiciona uma etapa de compilação ao fluxo de trabalho de desenvolvimento.

Comparativo Direto

Característica	JavaScript (JS)	TypeScript (TS)
Sistema de Tipos	Dinâmico e Fraco	Estático (opcional) e Forte (em compilação)
Verificação	Em tempo de execução	Em tempo de compilação
Deteção de Erros	Tardia (runtime)	Precoce (compile-time)
Coerção de Tipo	Implícita (frequente)	Explícita (geralmente exigida)
Curva de Aprendizado	Baixa (para iniciantes)	Média (adiciona conceitos de tipo)
Produtividade	Rápida para protótipos pequenos	Alta para projetos grandes e complexos
Ferramentas (IDE)	Básica	Avançada (autocompletar, refatoração segura)
Compatibilidade	Executa diretamente no navegador/Node.js	Requer compilação para JavaScript

Em resumo, enquanto JavaScript oferece flexibilidade e rapidez para pequenos scripts, TypeScript brilha em projetos maiores e mais complexos, onde a segurança de tipo e a manutenibilidade são cruciais. A escolha entre eles depende do tamanho do projeto, da equipe e dos requisitos de robustez.

Documentação Oficial:

- TypeScript: Handbook - Everyday Types
- TypeScript: Handbook - Type Inference

3. Estruturas de Controle

As estruturas de controle são blocos de código que permitem controlar o fluxo de execução de um programa, baseando-se em condições ou na repetição de tarefas. JavaScript, TypeScript e PHP compartilham muitas dessas estruturas, com pequenas variações sintáticas e de comportamento.

3.1. Estruturas Condicionais

if, else if, else Permitem que o código seja executado condicionalmente.

Sintaxe Geral:

```
1 if (condicao) {
2   // código a ser executado se a condição for verdadeira
3 } else if (outraCondicao) {
4   // código a ser executado se a outraCondicao for verdadeira
5 } else {
6   // código a ser executado se nenhuma das condições anteriores for verdadeira
7 }
```

JavaScript/TypeScript:

JavaScript:

```
1 let hora = 14;
2 if (hora < 12) {
3   console.log("Bom dia!");
4 } else if (hora < 18) {
5   console.log("Boa tarde!");
6 } else {
7   console.log("Boa noite!");
8 }
9 // Saída: Boa tarde!
10
11 let idade = 18;
12 if (idade >= 18) {
13   console.log("Você é maior de idade.");
14 } else {
15   console.log("Você é menor de idade.");
16 }
17 // Saída: Você é maior de idade.
```

TypeScript:

```
1 let horaTS: number = 14;
2 if (horaTS < 12) {
3   console.log("Bom dia!");
4 } else if (horaTS < 18) {
5   console.log("Boa tarde!");
6 } else {
7   console.log("Boa noite!");
8 }
9 // Saída: Boa tarde!
10
11 let idadeTS: number = 18;
12 if (idadeTS >= 18) {
13   console.log("Você é maior de idade.");
14 } else {
15   console.log("Você é menor de idade.");
16 }
```

```
17 // Saída: Você é maior de idade.
```

PHP:

```
1 $hora = 14;
2 if ($hora < 12) {
3     echo "Bom dia!\n";
4 } else if ($hora < 18) {
5     echo "Boa tarde!\n";
6 } else {
7     echo "Boa noite!\n";
8 }
9 // Saída: Boa tarde!
10
11 $idade = 18;
12 if ($idade >= 18) {
13     echo "Você é maior de idade.\n";
14 } else {
15     echo "Você é menor de idade.\n";
16 }
17 // Saída: Você é maior de idade.
```

Observações:

- As estruturas condicionais if, else if e else funcionam de forma idêntica nas três linguagens, com pequenas diferenças sintáticas (uso de \$ para variáveis em PHP).

Documentação Oficial:

- JavaScript: if...else
- PHP: if...else

switch Permite controlar o fluxo de execução com base em múltiplos valores possíveis de uma expressão.

Sintaxe Geral:

```
1 switch (expressao) {
2     case valor1:
3         // código a ser executado se expressao for igual a valor1
4         break;
5     case valor2:
6         // código a ser executado se expressao for igual a valor2
7         break;
8     default:
```

```
9 // código a ser executado se nenhum dos casos anteriores for correspondido
10 }
```

JavaScript/TypeScript:

JavaScript:

```
1 let diaDaSemana = 3;
2 let nomeDoDia;
3
4 switch (diaDaSemana) {
5   case 1:
6     nomeDoDia = "Domingo";
7     break;
8   case 2:
9     nomeDoDia = "Segunda-feira";
10    break;
11   case 3:
12     nomeDoDia = "Terça-feira";
13     break;
14   case 4:
15     nomeDoDia = "Quarta-feira";
16     break;
17   case 5:
18     nomeDoDia = "Quinta-feira";
19     break;
20   case 6:
21     nomeDoDia = "Sexta-feira";
22     break;
23   case 7:
24     nomeDoDia = "Sábado";
25     break;
26   default:
27     nomeDoDia = "Dia inválido";
28 }
29 console.log(nomeDoDia); // Terça-feira
```

TypeScript:

```
1 let diaDaSemanaTS: number = 3;
2 let nomeDoDiaTS: string;
3
4 switch (diaDaSemanaTS) {
5   case 1:
6     nomeDoDiaTS = "Domingo";
7     break;
8   case 2:
9     nomeDoDiaTS = "Segunda-feira";
```

```

10     break;
11 case 3:
12     nomeDoDiaTS = "Terça-feira";
13     break;
14 case 4:
15     nomeDoDiaTS = "Quarta-feira";
16     break;
17 case 5:
18     nomeDoDiaTS = "Quinta-feira";
19     break;
20 case 6:
21     nomeDoDiaTS = "Sexta-feira";
22     break;
23 case 7:
24     nomeDoDiaTS = "Sábado";
25     break;
26 default:
27     nomeDoDiaTS = "Dia inválido";
28 }
29 console.log(nomeDoDiaTS); // Terça-feira

```

PHP:

```

1 $diaDaSemana = 3;
2 $nomeDoDia;
3
4 switch ($diaDaSemana) {
5     case 1:
6         $nomeDoDia = "Domingo";
7         break;
8     case 2:
9         $nomeDoDia = "Segunda-feira";
10        break;
11    case 3:
12        $nomeDoDia = "Terça-feira";
13        break;
14    case 4:
15        $nomeDoDia = "Quarta-feira";
16        break;
17    case 5:
18        $nomeDoDia = "Quinta-feira";
19        break;
20    case 6:
21        $nomeDoDia = "Sexta-feira";
22        break;
23    case 7:
24        $nomeDoDia = "Sábado";

```

```

25     break;
26 default:
27     $nomeDoDia = "Dia inválido";
28 }
29 echo $nomeDoDia . "\n"; // Terça-feira

```

Observações:

- A estrutura `switch` funciona de forma idêntica nas três linguagens. É importante usar `break` para evitar a execução de blocos de código subsequentes (`fall-through`).

Documentação Oficial:

- JavaScript: [switch](#)
- PHP: [switch](#)

3.2. Estruturas de Repetição

for Executa um bloco de código um número específico de vezes.

Sintaxe Geral:

```

1 for (inicializacao; condicao; incremento) {
2     // código a ser executado
3 }

```

JavaScript/TypeScript:

JavaScript:

```

1 for (let i = 0; i < 5; i++) {
2     console.log("JavaScript: " + i);
3 }

```

TypeScript:

```

1 for (let i: number = 0; i < 5; i++) {
2     console.log("TypeScript: " + i);
3 }

```

PHP:

```

1 for ($i = 0; $i < 5; $i++) {
2     echo "PHP: " . $i . "\n";
3 }

```

Observações:

- O loop `for` funciona de forma idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: `for`
- PHP: `for`

while Executa um bloco de código enquanto uma condição especificada for verdadeira.

Sintaxe Geral:

```
1 while (condicao) {  
2   // código a ser executado  
3 }
```

JavaScript/TypeScript:

```
1 let i = 0;  
2 while (i < 5) {  
3   console.log("JavaScript: " + i);  
4   i++;  
5 }
```

TypeScript:

```
1 let iTS: number = 0;  
2 while (iTS < 5) {  
3   console.log("TypeScript: " + iTS);  
4   iTS++;  
5 }
```

PHP:

```
1 $i = 0;  
2 while ($i < 5) {  
3   echo "PHP: " . $i . "\n";  
4   $i++;  
5 }
```

Observações:

- O loop `while` funciona de forma idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: `while`
- PHP: `while`

do ... while Executa um bloco de código uma vez, e então repete o loop enquanto a condição especificada for verdadeira. Garante que o bloco de código seja executado pelo menos uma vez.

Sintaxe Geral:

```
1 do {  
2   // código a ser executado  
3 } while (condicao);
```

JavaScript/TypeScript:

```
1 let i = 0;  
2 do {  
3   console.log("JavaScript: " + i);  
4   i++;  
5 } while (i < 5);
```

TypeScript:

```
1 let iTS: number = 0;  
2 do {  
3   console.log("TypeScript: " + iTS);  
4   iTS++;  
5 } while (iTS < 5);
```

PHP:

```
1 $i = 0;  
2 do {  
3   echo "PHP: " . $i . "\n";  
4   $i++;  
5 } while ($i < 5);
```

Observações:

- O loop do ... while funciona de forma idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: do...while
- PHP: do...while

for ... of (JavaScript/TypeScript) e foreach (PHP) Iteram sobre coleções de dados.

JavaScript/TypeScript (for ... of):

```
1 const arrayJS = [1, 2, 3];
2 for (const elemento of arrayJS) {
3   console.log("JavaScript (for...of): " + elemento);
4 }
5
6 const stringJS = "hello";
7 for (const char of stringJS) {
8   console.log("JavaScript (for...of string): " + char);
9 }
```

TypeScript:

```
1 const arrayTS: number[] = [1, 2, 3];
2 for (const elemento of arrayTS) {
3   console.log("TypeScript (for...of): " + elemento);
4 }
5
6 const stringTS: string = "hello";
7 for (const char of stringTS) {
8   console.log("TypeScript (for...of string): " + char);
9 }
```

PHP (foreach):

```
1 $arrayPHP = [1, 2, 3];
2 foreach ($arrayPHP as $elemento) {
3   echo "PHP (foreach): " . $elemento . "\n";
4 }
5
6 $associativeArrayPHP = [
7   "a" => 1,
8   "b" => 2,
9   "c" => 3
10 ];
11 foreach ($associativeArrayPHP as $chave => $valor) {
12   echo "PHP (foreach associativo): " . $chave . " => " . $valor . "\n";
13 }
```

Observações:

- for ... of em JavaScript/TypeScript é usado para iterar sobre valores de objetos iteráveis (Arrays, Strings, Maps, Sets, etc.).
- foreach em PHP é usado para iterar sobre elementos de arrays e objetos. Pode ser usado para acessar apenas valores ou pares chave-valor.

Documentação Oficial:

- JavaScript: for...of
- PHP: foreach

for ... in (JavaScript/TypeScript) Itera sobre as propriedades enumeráveis de um objeto.

JavaScript/TypeScript:

```
1 const objetoJS = { a: 1, b: 2, c: 3 };
2 for (const chave in objetoJS) {
3   console.log(`JavaScript (for...in): ${chave}: ${objetoJS[chave]}`);
4 }
```

TypeScript:

```
1 const objetoTS: { [key: string]: number } = { a: 1, b: 2, c: 3 };
2 for (const chave in objetoTS) {
3   console.log(`TypeScript (for...in): ${chave}: ${objetoTS[chave]}`);
4 }
```

Observações:

- for ... in em JavaScript/TypeScript é usado para iterar sobre as chaves (nomes das propriedades) de um objeto. Não é recomendado para iterar sobre arrays, pois pode incluir propriedades herdadas e a ordem não é garantida.

Documentação Oficial:

- JavaScript: for...in

3.3. Controle de Fluxo

break Termina a execução do loop atual ou da instrução switch e transfere o controle para a instrução que segue o loop ou switch.

JavaScript/TypeScript:

```
1 for (let i = 0; i < 10; i++) {
2   if (i === 5) {
3     break; // Sai do loop quando i for 5
4   }
5   console.log("JavaScript (break): " + i);
6 }
7 // Saída: 0, 1, 2, 3, 4
```

TypeScript:

```
1 for (let i: number = 0; i < 10; i++) {  
2   if (i === 5) {  
3     break; // Sai do loop quando i for 5  
4   }  
5   console.log("TypeScript (break): " + i);  
6 }  
7 // Saída: 0, 1, 2, 3, 4
```

PHP:

```
1 for ($i = 0; $i < 10; $i++) {  
2   if ($i === 5) {  
3     break; // Sai do loop quando $i for 5  
4   }  
5   echo "PHP (break): " . $i . "\n";  
6 }  
7 // Saída: 0, 1, 2, 3, 4
```

Observações:

- O break funciona de forma idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: break
- PHP: break

continue Termina a execução da iteração atual do loop e continua a execução do loop na próxima iteração.

JavaScript/TypeScript:

```
1 for (let i = 0; i < 5; i++) {  
2   if (i === 2) {  
3     continue; // Pula a iteração quando i for 2  
4   }  
5   console.log("JavaScript (continue): " + i);  
6 }  
7 // Saída: 0, 1, 3, 4
```

TypeScript:

```

1 for (let i: number = 0; i < 5; i++) {
2   if (i === 2) {
3     continue; // Pula a iteração quando i for 2
4   }
5   console.log("TypeScript (continue): " + i);
6 }
7 // Saída: 0, 1, 3, 4

```

PHP:

```

1 for ($i = 0; $i < 5; $i++) {
2   if ($i === 2) {
3     continue; // Pula a iteração quando $i for 2
4   }
5   echo "PHP (continue): " . $i . "\n";
6 }
7 // Saída: 0, 1, 3, 4

```

Observações:

- O `continue` funciona de forma idêntica nas três linguagens.

Documentação Oficial:

- JavaScript: `continue`
- PHP: `continue`

4. Programação Orientada a Objetos (POO)

Conceitos Fundamentais da POO (Classes, Objetos, Atributos, Métodos)

Classes Uma classe é um molde ou um projeto para criar objetos. Ela define as propriedades (atributos) e os comportamentos (métodos) que os objetos criados a partir dela terão.

JavaScript/TypeScript:

```

1 class Pessoa {
2   constructor(nome, idade) {
3     this.nome = nome;
4     this.idade = idade;
5   }
6
7   apresentar() {
8     console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
9   }
10 }

```

TypeScript:

```
1 class PessoaTS {
2   nome: string;
3   idade: number;
4
5   constructor(nome: string, idade: number) {
6     this.nome = nome;
7     this.idade = idade;
8   }
9
10  apresentar(): void {
11    console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
12  }
13 }
```

PHP:

```
1 class Pessoa {
2   public $nome;
3   public $idade;
4
5   public function __construct($nome, $idade) {
6     $this->nome = $nome;
7     $this->idade = $idade;
8   }
9
10  public function apresentar() {
11    echo "Olá, meu nome é " . $this->nome . " e tenho " . $this->idade . " anos.\n";
12  }
13 }
```

Observações:

- Tanto JavaScript/TypeScript quanto PHP utilizam a palavra-chave `class` para definir classes. Em PHP, é necessário declarar a visibilidade (`public`, `private`, `protected`) dos atributos e métodos.

Objetos Um objeto é uma instância de uma classe. É uma entidade concreta criada a partir do molde definido pela classe.

JavaScript/TypeScript:

```
1 const pessoa1 = new Pessoa("Alice", 30);
2 pessoa1.apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

TypeScript:

```
1 const pessoa1TS: PessoaTS = new PessoaTS("Alice", 30);
2 pessoa1TS.apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

PHP:

```
1 $pessoa1 = new Pessoa("Alice", 30);
2 $pessoa1->apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

Observações:

- A criação de objetos é similar em ambas as linguagens, utilizando a palavra-chave `new`.

Atributos (Propriedades) São as características ou dados que um objeto possui. Representam o estado do objeto.

JavaScript/TypeScript:

```
1 // Exemplo na classe Pessoa: this.nome e this.idade são atributos
2 console.log(pessoa1.nome); // Alice
3 console.log(pessoa1.idade); // 30
```

TypeScript:

```
1 // Exemplo na classe PessoaTS: this.nome e this.idade são atributos
2 console.log(pessoa1TS.nome); // Alice
3 console.log(pessoa1TS.idade); // 30
```

PHP:

```
1 // Exemplo na classe Pessoa: $this->nome e $this->idade são atributos
2 echo $pessoa1->nome . "\n"; // Alice
3 echo $pessoa1->idade . "\n"; // 30
```

Observações:

- O acesso aos atributos é feito de forma similar, usando a notação de ponto (.) em JavaScript/TypeScript e a notação de seta (->) em PHP.

Métodos São as ações ou comportamentos que um objeto pode realizar. Representam a funcionalidade do objeto.

JavaScript/TypeScript:

```
1 // Exemplo na classe Pessoa: apresentar() é um método
2 pessoa1.apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

TypeScript:


```
1 // Exemplo na classe PessoaTS: apresentar() é um método
2 pessoa1TS.apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

PHP:

```
1 // Exemplo na classe Pessoa: apresentar() é um método
2 $pessoa1->apresentar(); // Olá, meu nome é Alice e tenho 30 anos.
```

Observações:

- A chamada de métodos é similar em ambas as linguagens, usando a notação de ponto (.) em JavaScript/TypeScript e a notação de seta (->) em PHP.

Encapsulamento

Encapsulamento é o princípio de agrupar dados (atributos) e os métodos que operam nesses dados em uma única unidade (a classe), e restringir o acesso direto a alguns dos componentes do objeto. Isso protege a integridade dos dados e permite que a implementação interna da classe seja alterada sem afetar o código externo que a utiliza.

JavaScript/TypeScript:

Em JavaScript, o encapsulamento tradicional (com modificadores de acesso `public`, `private`, `protected`) não existe nativamente da mesma forma que em linguagens como Java ou PHP. No entanto, existem convenções e recursos que permitem simular o encapsulamento:

- **Convenção:** Usar um prefixo `_` (underscore) para indicar que um atributo ou método é privado ou protegido. Isso é apenas uma convenção e não impede o acesso direto.
- **Closures:** Utilizar closures para criar variáveis e funções privadas.
- **Campos de Classe Privados (ES2019+):** Com a introdução de campos de classe privados (`#`), JavaScript agora oferece um mecanismo nativo para encapsulamento real.

```
1 class ContaBancaria {
2   #saldo; // Campo de classe privado
3
4   constructor(saldoInicial) {
5     this.#saldo = saldoInicial;
6   }
7
8   depositar(valor) {
9     if (valor > 0) {
```

```

10     this.#saldo += valor;
11     console.log(`Depósito de ${valor} realizado. Novo saldo: ${this.#saldo}`);
12 } else {
13     console.log("Valor de depósito inválido.");
14 }
15 }
16
17 sacar(valor) {
18     if (valor > 0 && valor <= this.#saldo) {
19         this.#saldo -= valor;
20         console.log(`Saque de ${valor} realizado. Novo saldo: ${this.#saldo}`);
21     } else {
22         console.log("Saldo insuficiente ou valor de saque inválido.");
23     }
24 }
25
26 getSaldo() {
27     return this.#saldo;
28 }
29 }
30
31 const minhaConta = new ContaBancaria(1000);
32 minhaConta.depositar(200);
33 minhaConta.sacar(500);
34 // console.log(minhaConta.#saldo); // Erro: Campo privado não pode ser acessado diretamente
35 console.log(`Saldo atual: ${minhaConta.getSaldo()}`);

```

TypeScript:

```

1 class ContaBancariaTS {
2     private _saldo: number; // Modificador private em TypeScript
3
4     constructor(saldoInicial: number) {
5         this._saldo = saldoInicial;
6     }
7
8     depositar(valor: number): void {
9         if (valor > 0) {
10             this._saldo += valor;
11             console.log(`Depósito de ${valor} realizado. Novo saldo: ${this._saldo}`);
12         } else {
13             console.log("Valor de depósito inválido.");
14         }
15     }
16
17     sacar(valor: number): void {
18         if (valor > 0 && valor <= this._saldo) {

```

```

19     this._saldo -= valor;
20     console.log(`Saque de ${valor} realizado. Novo saldo: ${this._saldo}`);
21   } else {
22     console.log("Saldo insuficiente ou valor de saque inválido.");
23   }
24 }
25
26 getSaldo(): number {
27   return this._saldo;
28 }
29 }
30
31 const minhaContaTS = new ContaBancariaTS(1000);
32 minhaContaTS.depositar(200);
33 minhaContaTS.sacar(500);
34 // console.log(minhaContaTS._saldo); // Erro em tempo de compilação: Propriedade privada
35 console.log(`Saldo atual: ${minhaContaTS.getSaldo()}`);

```

TypeScript:

TypeScript, além dos campos de classe privados do JavaScript, oferece modificadores de acesso (`public`, `private`, `protected`) que são aplicados em tempo de compilação para garantir o encapsulamento. No entanto, após a compilação para JavaScript, esses modificadores não existem mais, e o encapsulamento real depende dos recursos do JavaScript (como os campos privados).

```

1 class ContaBancariaTS {
2   private _saldo: number; // Modificador private em TypeScript
3
4   constructor(saldoInicial: number) {
5     this._saldo = saldoInicial;
6   }
7
8   depositar(valor: number): void {
9     if (valor > 0) {
10      this._saldo += valor;
11      console.log(`Depósito de ${valor} realizado. Novo saldo: ${this._saldo}`);
12    } else {
13      console.log("Valor de depósito inválido.");
14    }
15  }
16
17  sacar(valor: number): void {
18    if (valor > 0 && valor <= this._saldo) {
19      this._saldo -= valor;
20      console.log(`Saque de ${valor} realizado. Novo saldo: ${this._saldo}`);

```

```

21     } else {
22         console.log("Saldo insuficiente ou valor de saque inválido.");
23     }
24 }
25
26 getSaldo(): number {
27     return this._saldo;
28 }
29 }
30
31 const minhaContaTS = new ContaBancariaTS(1000);
32 minhaContaTS.depositar(200);
33 minhaContaTS.sacar(500);
34 // console.log(minhaContaTS._saldo); // Erro em tempo de compilação: Propriedade privada
35 console.log(`Saldo atual: ${minhaContaTS.getSaldo()}`);

```

PHP:

PHP possui modificadores de acesso explícitos (`public`, `private`, `protected`) que controlam a visibilidade de propriedades e métodos.

- `public`: A propriedade ou método pode ser acessado de qualquer lugar.
- `private`: A propriedade ou método só pode ser acessado de dentro da própria classe.
- `protected`: A propriedade ou método pode ser acessado de dentro da própria classe e de classes que herdam dela.

```

1 class ContaBancariaPHP {
2     private $saldo; // Propriedade privada
3
4     public function __construct($saldoInicial) {
5         $this->saldo = $saldoInicial;
6     }
7
8     public function depositar($valor) {
9         if ($valor > 0) {
10             $this->saldo += $valor;
11             echo "Depósito de " . $valor . " realizado. Novo saldo: " . $this->saldo . "\n";
12         } else {
13             echo "Valor de depósito inválido.\n";
14         }
15     }
16
17     public function sacar($valor) {
18         if ($valor > 0 && $valor <= $this->saldo) {

```

```

19     $this->saldo -= $valor;
20     echo "Saque de " . $valor . " realizado. Novo saldo: " . $this->saldo . "\n";
21 } else {
22     echo "Saldo insuficiente ou valor de saque inválido.\n";
23 }
24 }
25
26 public function getSaldo() {
27     return $this->saldo;
28 }
29 }
30
31 $minhaContaPHP = new ContaBancariaPHP(1000);
32 $minhaContaPHP->depositar(200);
33 $minhaContaPHP->sacar(500);
34 // echo $minhaContaPHP->saldo; // Erro fatal: Não é possível acessar propriedade privada
35 echo "Saldo atual: " . $minhaContaPHP->getSaldo() . "\n";

```

Observações:

- O encapsulamento é fundamental para a segurança e manutenção do código, garantindo que o estado interno de um objeto seja manipulado apenas por seus próprios métodos.

Documentação Oficial:

- JavaScript: Campos de Classe Privados
- TypeScript: Modificadores de Acesso
- PHP: Visibilidade

Herança

Herança é um mecanismo que permite que uma classe (subclasse ou classe filha) herde propriedades e métodos de outra classe (superclasse ou classe pai). Isso promove a reutilização de código e estabelece uma relação “é um tipo de” entre as classes.

JavaScript/TypeScript:

Utilizam a palavra-chave `extends` para herança.

```

1 class Animal {
2     constructor(nome) {
3         this.nome = nome;
4     }

```

```

5
6  fazerBarulho() {
7      console.log("Algum barulho...");
8  }
9  }
10
11 class Cachorro extends Animal {
12     constructor(nome, raca) {
13         super(nome);
14         this.raca = raca;
15     }
16
17     fazerBarulho() {
18         console.log("Au au!");
19     }
20
21     latir() {
22         console.log("O cachorro " + this.nome + " está latindo!");
23     }
24 }
25
26 const meuCachorro = new Cachorro("Rex", "Labrador");
27 meuCachorro.fazerBarulho(); // Au au!
28 meuCachorro.latir();       // O cachorro Rex está latindo!
29 console.log(meuCachorro.nome); // Rex

```

TypeScript:

```

1 class AnimalTSHeranca {
2     constructor(protected nome: string) {}
3
4     fazerBarulho(): void {
5         console.log("Algum barulho...");
6     }
7 }
8
9 class CachorroTSHeranca extends AnimalTSHeranca {
10     constructor(nome: string, public raca: string) {
11         super(nome); // Chama o construtor da classe pai
12     }
13
14     fazerBarulho(): void {
15         console.log("Au au!");
16     }
17
18     latir(): void {
19         console.log(`O cachorro ${this.nome} está latindo!`);

```

```

20 }
21 }
22
23 const meuCachorroTSHeranca: CachorroTSHeranca = new CachorroTSHeranca("Rex", "Labrador");
24 meuCachorroTSHeranca.fazerBarulho(); // Au au!
25 meuCachorroTSHeranca.latir();          // O cachorro Rex está latindo!
26 console.log(meuCachorroTSHeranca.nome); // Rex

```

PHP:

Também utiliza a palavra-chave `extends` para herança.

```

1 class Animal {
2     public $nome;
3
4     public function __construct($nome) {
5         $this->nome = $nome;
6     }
7
8     public function fazerBarulho() {
9         echo "Algum barulho...\n";
10    }
11 }
12
13 class Cachorro extends Animal {
14     public $raca;
15
16     public function __construct($nome, $raca) {
17         parent::__construct($nome); // Chama o construtor da classe pai
18         $this->raca = $raca;
19     }
20
21     public function fazerBarulho() {
22         echo "Au au!\n";
23     }
24
25     public function latir() {
26         echo "O cachorro " . $this->nome . " está latindo!\n";
27     }
28 }
29
30 $meuCachorro = new Cachorro("Rex", "Labrador");
31 $meuCachorro->fazerBarulho(); // Au au!
32 $meuCachorro->latir();        // O cachorro Rex está latindo!
33 echo $meuCachorro->nome . "\n"; // Rex

```

Observações:

- Ambas as linguagens usam `extends` para herdar. Em JavaScript/TypeScript, `super()` é usado para chamar o construtor da classe pai. Em PHP, `parent::__construct()` é usado para o mesmo propósito.
- Métodos podem ser sobrescritos nas subclasses para fornecer implementações específicas.

Documentação Oficial:

- JavaScript: `extends`
- PHP: Herança de Objetos

Polimorfismo

Polimorfismo significa “muitas formas”. Em POO, refere-se à capacidade de objetos de diferentes classes responderem ao mesmo método de maneiras diferentes, desde que essas classes compartilhem uma interface comum ou herdem de uma superclasse comum. Isso permite que o código seja mais flexível e extensível.

JavaScript/TypeScript:

O polimorfismo é naturalmente suportado através da herança e da sobrescrita de métodos.

```
1 class AnimalGenerico {
2   fazerBarulho() {
3     console.log("Barulho genérico de animal.");
4   }
5 }
6
7 class Gato extends AnimalGenerico {
8   fazerBarulho() {
9     console.log("Miau!");
10  }
11 }
12
13 class Pato extends AnimalGenerico {
14   fazerBarulho() {
15     console.log("Quack!");
16   }
17 }
18
19 function emitirBarulho(animal) {
20   animal.fazerBarulho();
21 }
22
23 const meuGato = new Gato();
```



```

24 const meuPato = new Pato();
25 const meuAnimalGenerico = new AnimalGenerico();
26
27 emitirBarulho(meuGato);           // Miau!
28 emitirBarulho(meuPato);           // Quack!
29 emitirBarulho(meuAnimalGenerico); // Barulho genérico de animal.

```

TypeScript:

```

1 class AnimalGenericoTS {
2     fazerBarulho(): void {
3         console.log("Barulho genérico de animal.");
4     }
5 }
6
7 class GatoTS extends AnimalGenericoTS {
8     fazerBarulho(): void {
9         console.log("Miau!");
10    }
11 }
12
13 class PatoTS extends AnimalGenericoTS {
14     fazerBarulho(): void {
15         console.log("Quack!");
16     }
17 }
18
19 function emitirBarulhoTS(animal: AnimalGenericoTS) {
20     animal.fazerBarulho();
21 }
22
23 const meuGatoTS: GatoTS = new GatoTS();
24 const meuPatoTS: PatoTS = new PatoTS();
25 const meuAnimalGenericoTS: AnimalGenericoTS = new AnimalGenericoTS();
26
27 emitirBarulhoTS(meuGatoTS);           // Miau!
28 emitirBarulhoTS(meuPatoTS);           // Quack!
29 emitirBarulhoTS(meuAnimalGenericoTS); // Barulho genérico de animal.

```

PHP:

O polimorfismo também é suportado através da herança e da sobrescrita de métodos, e pode ser reforçado com interfaces.

```

1 class AnimalGenericoPHP {
2     public function fazerBarulho() {
3         echo "Barulho genérico de animal.\n";
4     }

```

```

5 }
6
7 class GatoPHP extends AnimalGenericoPHP {
8     public function fazerBarulho() {
9         echo "Miau!\n";
10    }
11 }
12
13 class PatoPHP extends AnimalGenericoPHP {
14     public function fazerBarulho() {
15         echo "Quack!\n";
16    }
17 }
18
19 function emitirBarulhoPHP(AnimalGenericoPHP $animal) {
20     $animal->fazerBarulho();
21 }
22
23 $meuGatoPHP = new GatoPHP();
24 $meuPatoPHP = new PatoPHP();
25 $meuAnimalGenericoPHP = new AnimalGenericoPHP();
26
27 emitirBarulhoPHP($meuGatoPHP);           // Miau!
28 emitirBarulhoPHP($meuPatoPHP);           // Quack!
29 emitirBarulhoPHP($meuAnimalGenericoPHP); // Barulho genérico de animal.

```

Observações:

- O polimorfismo permite que você escreva código mais genérico que pode operar em objetos de diferentes tipos, desde que eles compartilhem uma base comum (herança ou interface).

Documentação Oficial:

- JavaScript: Polimorfismo (conceito)
- PHP: Polimorfismo (conceito)

Interfaces e Classes Abstratas (onde aplicável)

Interfaces Interfaces definem um contrato que as classes devem seguir. Elas especificam quais métodos uma classe deve implementar, mas não fornecem a implementação desses métodos. Uma classe pode implementar várias interfaces.

TypeScript:

TypeScript possui o conceito de interfaces para definir a forma de objetos e classes. Em tempo de execução, as interfaces são removidas, pois JavaScript não tem interfaces nativas.

```
1 interface FormaGeometrica {
2   calcularArea(): number;
3   calcularPerimetro(): number;
4 }
5
6 class Circulo implements FormaGeometrica {
7   constructor(private raio: number) {}
8
9   calcularArea(): number {
10    return Math.PI * this.raio * this.raio;
11  }
12
13  calcularPerimetro(): number {
14    return 2 * Math.PI * this.raio;
15  }
16 }
17
18 class Retangulo implements FormaGeometrica {
19   constructor(private largura: number, private altura: number) {}
20
21   calcularArea(): number {
22    return this.largura * this.altura;
23  }
24
25  calcularPerimetro(): number {
26    return 2 * (this.largura + this.altura);
27  }
28 }
29
30 const meuCirculo = new Circulo(5);
31 console.log(`Área do Círculo: ${meuCirculo.calcularArea()}`);
32 console.log(`Perímetro do Círculo: ${meuCirculo.calcularPerimetro()}`);
33
34 const meuRetangulo = new Retangulo(4, 6);
35 console.log(`Área do Retângulo: ${meuRetangulo.calcularArea()}`);
36 console.log(`Perímetro do Retângulo: ${meuRetangulo.calcularPerimetro()}`);
```

PHP:

PHP possui interfaces nativas que são usadas para definir contratos para classes.

```
1 interface FormaGeometricaPHP {
2   public function calcularArea(): float;
```

```

3  public function calcularPerimetro(): float;
4  }
5
6  class CirculoPHP implements FormaGeometricaPHP {
7      private $raio;
8
9      public function __construct(float $raio) {
10         $this->raio = $raio;
11     }
12
13     public function calcularArea(): float {
14         return M_PI * $this->raio * $this->raio;
15     }
16
17     public function calcularPerimetro(): float {
18         return 2 * M_PI * $this->raio;
19     }
20 }
21
22 class RetanguloPHP implements FormaGeometricaPHP {
23     private $largura;
24     private $altura;
25
26     public function __construct(float $largura, float $altura) {
27         $this->largura = $largura;
28         $this->altura = $altura;
29     }
30
31     public function calcularArea(): float {
32         return $this->largura * $this->altura;
33     }
34
35     public function calcularPerimetro(): float {
36         return 2 * ($this->largura + $this->altura);
37     }
38 }
39
40 $meuCirculoPHP = new CirculoPHP(5);
41 echo "Área do Círculo: " . $meuCirculoPHP->calcularArea() . "\n";
42 echo "Perímetro do Círculo: " . $meuCirculoPHP->calcularPerimetro() . "\n";
43
44 $meuRetanguloPHP = new RetanguloPHP(4, 6);
45 echo "Área do Retângulo: " . $meuRetanguloPHP->calcularArea() . "\n";
46 echo "Perímetro do Retângulo: " . $meuRetanguloPHP->calcularPerimetro() . "\n";

```

Observações:

- Interfaces são úteis para definir contratos e garantir que as classes que as implementam forneçam métodos específicos.

Documentação Oficial:

- TypeScript: Interfaces
- PHP: Interfaces de Objeto

Classes Abstratas Classes abstratas são classes que não podem ser instanciadas diretamente e podem conter métodos abstratos (métodos sem implementação) e métodos concretos (métodos com implementação). Elas servem como base para outras classes, que devem implementar os métodos abstratos.

TypeScript:

TypeScript suporta classes abstratas.

```
1 abstract class AnimalAbstrato {
2     constructor(protected nome: string) {}
3
4     abstract fazerBarulho(): void; // Método abstrato
5
6     mover(): void {
7         console.log(`${this.nome} está se movendo.`);
8     }
9 }
10
11 class CachorroConcreto extends AnimalAbstrato {
12     constructor(nome: string, private raca: string) {
13         super(nome);
14     }
15
16     fazerBarulho(): void {
17         console.log("Au au!");
18     }
19
20     mostrarRaca(): void {
21         console.log(`A raça do ${this.nome} é ${this.raca}.`);
22     }
23 }
24
25 // const animal = new AnimalAbstrato("Bicho"); // Erro: Não é possível instanciar uma
    classe abstrata
26
27 const meuCachorroConcreto = new CachorroConcreto("Buddy", "Golden Retriever");
```

```

28 meuCachorroConcreto.fazerBarulho(); // Au au!
29 meuCachorroConcreto.mover();        // Buddy está se movendo.
30 meuCachorroConcreto.mostrarRaca();   // A raça do Buddy é Golden Retriever.

```

PHP:

PHP também suporta classes abstratas.

```

1  abstract class AnimalAbstratoPHP {
2      protected $nome;
3
4      public function __construct($nome) {
5          $this->nome = $nome;
6      }
7
8      abstract public function fazerBarulho(); // Método abstrato
9
10     public function mover() {
11         echo $this->nome . " está se movendo.\n";
12     }
13 }
14
15 class CachorroConcretoPHP extends AnimalAbstratoPHP {
16     private $raca;
17
18     public function __construct($nome, $raca) {
19         parent::__construct($nome);
20         $this->raca = $raca;
21     }
22
23     public function fazerBarulho() {
24         echo "Au au!\n";
25     }
26
27     public function mostrarRaca() {
28         echo "A raça do " . $this->nome . " é " . $this->raca . ".\n";
29     }
30 }
31
32 // $animalPHP = new AnimalAbstratoPHP("Bicho"); // Erro fatal: Não é possível instanciar
    uma classe abstrata
33
34 $meuCachorroConcretoPHP = new CachorroConcretoPHP("Buddy", "Golden Retriever");
35 $meuCachorroConcretoPHP->fazerBarulho(); // Au au!
36 $meuCachorroConcretoPHP->mover();        // Buddy está se movendo.
37 $meuCachorroConcretoPHP->mostrarRaca();   // A raça do Buddy é Golden Retriever.

```

Observações:

- Classes abstratas são usadas para definir uma estrutura comum para um conjunto de classes relacionadas, forçando as subclasses a implementar certos métodos.

Documentação Oficial:

- TypeScript: Classes Abstratas
- PHP: Classes Abstratas

5. Referencias Usadas

Autor: Manus AI

Referências:

[1] MDN Web Docs - JavaScript [2] TypeScript Documentation [3] Manual do PHP [4] JavaScript: Operadores Aritméticos [5] PHP: Operadores Aritméticos [6] JavaScript: Operadores de Atribuição [7] PHP: Operadores de Atribuição [8] JavaScript: Operadores de Comparação [9] PHP: Operadores de Comparação [10] JavaScript: Operadores Lógicos [11] PHP: Operadores Lógicos [12] JavaScript: Operador de Concatenação [13] PHP: Operadores de String [14] JavaScript: Operadores de Incremento e Decremento [15] PHP: Operadores de Incremento/Decremento [16] JavaScript: Operador Condicional (Ternário) [17] PHP: Operador Ternário [18] JavaScript: Operador typeof [19] JavaScript: Operador instanceof [20] PHP: Funções de Tipo [21] PHP: Operador instanceof [22] JavaScript: if...else [23] PHP: if...else [24] JavaScript: switch [25] PHP: switch [26] JavaScript: for [27] PHP: for [28] JavaScript: while [29] PHP: while [30] JavaScript: do...while [31] PHP: do...while [32] JavaScript: for...of [33] PHP: foreach [34] JavaScript: for...in [35] JavaScript: break [36] PHP: break [37] JavaScript: continue [38] PHP: continue [39] JavaScript: Campos de Classe Privados [40] TypeScript: Modificadores de Acesso [41] PHP: Visibilidade [42] JavaScript: extends [43] PHP: Herança de Objetos [44] JavaScript: Polimorfismo (conceito) [45] PHP: Polimorfismo (conceito) [46] TypeScript: Interfaces [47] PHP: Interfaces de Objeto [48] TypeScript: Classes Abstratas [49] PHP: Classes Abstratas

Orientação a Objetos por Prototipação (JavaScript)

Ao contrário de linguagens como Java, C++ ou PHP, que são baseadas em classes, o JavaScript é uma linguagem baseada em protótipos. Isso significa que, em vez de classes, o

JavaScript utiliza objetos existentes (protótipos) como modelos para criar novos objetos. A herança é alcançada através da cadeia de protótipos, onde um objeto pode herdar propriedades e métodos de outro objeto.

Mesmo com a introdução da sintaxe `class` no ES2015 (ES6), é importante entender que essa é apenas uma “açúcar sintático” (syntactic sugar) sobre o modelo de protótipos existente. Por baixo dos panos, o JavaScript ainda utiliza protótipos para implementar classes e herança.

Como funciona a Prototipação Cada objeto em JavaScript tem uma propriedade interna `[[Prototype]]` (acessível via `__proto__` em navegadores, ou `Object.getPrototypeOf()`), que aponta para outro objeto, seu protótipo. Quando você tenta acessar uma propriedade ou método em um objeto, e ele não é encontrado diretamente nesse objeto, o JavaScript procura na cadeia de protótipos até encontrar a propriedade ou método ou chegar ao final da cadeia (`null`).

```
1 // Objeto protótipo para Animais
2 const AnimalProto = {
3   fazerBarulho: function() {
4     console.log("Barulho genérico de animal.");
5   },
6   comer: function() {
7     console.log(this.nome + " está comendo.");
8   }
9 };
10
11 // Função construtora para criar objetos Cachorro
12 function Cachorro(nome, raca) {
13   this.nome = nome;
14   this.raca = raca;
15 }
16
17 // Definindo o protótipo de Cachorro para herdar de AnimalProto
18 // Object.create cria um novo objeto com o protótipo especificado
19 Cachorro.prototype = Object.create(AnimalProto);
20 Cachorro.prototype.constructor = Cachorro; // Corrige o construtor
21
22 // Adicionando um método específico para Cachorro
23 Cachorro.prototype.latir = function() {
24   console.log(this.nome + " está latindo: Au au!");
25 };
26
27 const meuCachorro = new Cachorro("Rex", "Labrador");
28
29 meuCachorro.fazerBarulho(); // Saída: Barulho genérico de animal.
```



```

30 meuCachorro.comer();           // Saída: Rex está comendo.
31 meuCachorro.latir();           // Saída: Rex está latindo: Au au!
32
33 console.log(meuCachorro.__proto__ === Cachorro.prototype); // true
34 console.log(Cachorro.prototype.__proto__ === AnimalProto); // true
35 console.log(AnimalProto.__proto__ === Object.prototype);   // true
36 console.log(Object.prototype.__proto__ === null);           // true

```

Neste exemplo:

- `AnimalProto` é um objeto simples que serve como protótipo para `Cachorro`.
- `Cachorro.prototype` é definido para herdar de `AnimalProto` usando `Object.create()`. Isso estabelece a cadeia de protótipos.
- Quando `meuCachorro.fazerBarulho()` é chamado, o JavaScript primeiro procura `fazerBarulho` em `meuCachorro`. Não encontrando, ele procura em `Cachorro.prototype`. Não encontrando, ele procura em `AnimalProto`, onde encontra e executa o método.

Prototipação com a sintaxe `class` (ES6+) Como mencionado, a sintaxe `class` é uma abstração sobre o modelo de protótipos. O exemplo de herança com `class` que você já tem no material (`class Animal` e `class Cachorro extends Animal`) internamente utiliza a cadeia de protótipos para funcionar. Quando você define `class Cachorro extends Animal`, o JavaScript faz o seguinte por baixo dos panos:

- Define `Cachorro.prototype` para herdar de `Animal.prototype`.
- Garante que `super()` no construtor de `Cachorro` chame o construtor de `Animal` e configure corretamente o `this`.

```

1 // Relembrando o exemplo com 'class'
2 class Animal {
3   constructor(nome) {
4     this.nome = nome;
5   }
6
7   fazerBarulho() {
8     console.log("Algum barulho...");
9   }

```

```

10 }
11
12 class Cachorro extends Animal {
13     constructor(nome, raca) {
14         super(nome);
15         this.raca = raca;
16     }
17
18     fazerBarulho() {
19         console.log("Au au!");
20     }
21
22     latir() {
23         console.log("O cachorro " + this.nome + " está latindo!");
24     }
25 }
26
27 const meuCachorroES6 = new Cachorro("Buddy", "Poodle");
28 meuCachorroES6.fazerBarulho(); // Au au!
29 meuCachorroES6.latir();        // O cachorro Buddy está latindo!
30
31 // A cadeia de protótipos ainda está lá!
32 console.log(meuCachorroES6.__proto__ === Cachorro.prototype); // true
33 console.log(Cachorro.prototype.__proto__ === Animal.prototype); // true
34 console.log(Animal.prototype.__proto__ === Object.prototype); // true

```

Observações:

- Entender a prototipação é crucial para dominar o JavaScript, mesmo usando a sintaxe `class`. Problemas como o uso incorreto de `this` em funções e a otimização de memória em aplicações JavaScript muitas vezes se relacionam diretamente com o modelo de protótipos.
- A prototipação permite uma herança mais flexível e dinâmica do que a herança baseada em classes, possibilitando a modificação de protótipos em tempo de execução.

Documentação Oficial:

- JavaScript: Herança e cadeia de protótipos
- JavaScript: Classes

Recursos Adicionais:

6. Introdução a Laravel, React Native e SQL

6.2. Laravel

Laravel é um framework PHP de código aberto, robusto e elegante, projetado para o desenvolvimento de aplicações web modernas. Ele segue o padrão de arquitetura Model-View-Controller (MVC) e oferece uma vasta gama de ferramentas e recursos que simplificam tarefas comuns de desenvolvimento, como roteamento, autenticação, ORM (Object-Relational Mapping), filas, e muito mais. A filosofia do Laravel é tornar o desenvolvimento web uma experiência agradável e produtiva.

Principais Conceitos e Comandos (Laravel)

Instalação e Configuração Para começar com Laravel, você precisa ter PHP e Composer (gerenciador de dependências do PHP) instalados. O Laravel Installer é a maneira mais rápida de criar novos projetos.

- **Instalar Laravel Installer (globalmente):**
- **Criar um novo projeto Laravel:**
- **Iniciar o servidor de desenvolvimento:**

Artisan Console Artisan é a interface de linha de comando incluída no Laravel. Ele fornece uma série de comandos úteis para o desenvolvimento da sua aplicação.

- **Listar todos os comandos Artisan disponíveis:**
- **Obter ajuda para um comando específico:**

Rotas (Routing) As rotas definem como as requisições HTTP são tratadas pela sua aplicação. As rotas são definidas no arquivo `routes/web.php` (para rotas web) e `routes/api.php` (para APIs).

```
1 // routes/web.php
2
3 use Illuminate\Support\Facades\Route;
4
5 Route::get('/', function () {
6     return view('welcome');
```

```

7 });
8
9 Route::get('/saudacao/{nome}', function ($nome) {
10     return 'Olá, ' . $nome . '!';
11 });
12
13 Route::post('/contato', function () {
14     // Lógica para processar o formulário de contato
15     return 'Mensagem enviada!';
16 });

```

Controladores (Controllers) Controladores agrupam a lógica de requisição HTTP. Eles são criados usando o Artisan.

- **Criar um controlador:**
- **Exemplo de controlador (app/Http/Controllers/PostController.php):**
- **Associar rotas a controladores:**

Migrações (Migrations) Migrações são como controle de versão para o seu banco de dados, permitindo que você defina e modifique a estrutura do banco de dados usando código PHP.

- **Criar uma migração:**
- **Executar migrações (criar tabelas no banco de dados):**
- **Reverter a última migração:**

Modelos (Models) e Eloquent ORM Eloquent ORM é o ORM (Object-Relational Mapping) padrão do Laravel, que facilita a interação com o banco de dados usando objetos PHP.

- **Criar um modelo:**
- **Exemplo de modelo (app/Models/Post.php):**
- **Interagindo com o banco de dados usando Eloquent:**

Views (Blade) Blade é o motor de template simples, mas poderoso, incluído no Laravel. As views são armazenadas em `resources/views/`.

- **Exemplo de view (`resources/views/welcome.blade.php`):**
- **Retornar uma view de uma rota ou controlador:**

Observações:

- Laravel é um ecossistema vasto. Esta seção aborda apenas os conceitos fundamentais para iniciar o desenvolvimento. Recursos como autenticação, autorização, filas, eventos, testes e pacotes são componentes poderosos que o Laravel oferece.

Documentação Oficial:

- [Laravel: Documentação Oficial](#)

6.3. React Native

React Native é um framework de código aberto, criado pelo Facebook, que permite desenvolver aplicações móveis nativas para iOS e Android usando JavaScript e React. Ao invés de renderizar componentes web (como no React para web), o React Native renderiza componentes nativos da interface do usuário, proporcionando uma experiência de usuário e performance próximas às de aplicações desenvolvidas com linguagens nativas (Swift/Objective-C para iOS, Java/Kotlin para Android).

Principais Conceitos e Comandos (React Native)

Instalação e Configuração Existem duas maneiras principais de iniciar um projeto React Native:

1. **Expo Go (Recomendado para iniciantes e prototipagem rápida):** Permite desenvolver rapidamente sem a necessidade de configurar um ambiente de desenvolvimento nativo completo. Você escreve o código JavaScript e o Expo Go cuida da compilação e execução no seu dispositivo ou emulador.
- **Instalar Expo CLI (globalmente):**
 - **Criar um novo projeto Expo:**

- **Iniciar o servidor de desenvolvimento Expo:**

1. **React Native CLI (Para projetos mais complexos e controle total sobre o código nativo):** Requer a configuração de ambientes de desenvolvimento para iOS (Xcode) e Android (Android Studio).

- **Instalar React Native CLI (globalmente):**

- **Criar um novo projeto React Native:**

- **Executar o app no iOS (requer Xcode e macOS):**

- **Executar o app no Android (requer Android Studio e emulador/dispositivo):**

Componentes Essenciais React Native fornece um conjunto de componentes nativos que mapeiam para os elementos da UI nativa da plataforma. Alguns dos mais comuns incluem:

- **View:** O contêiner mais fundamental para construir a UI. Equivale a uma `div` no HTML.
- **Text:** Usado para exibir texto.
- **Image:** Usado para exibir imagens.
- **TextInput:** Um componente de entrada de texto.
- **Button:** Um componente de botão básico.
- **ScrollView:** Um contêiner que permite rolagem.
- **FlatList:** Um componente otimizado para exibir listas grandes de dados.
- **StyleSheet:** Usado para criar estilos para os componentes, similar ao CSS, mas com sintaxe JavaScript.

Exemplo Básico de Componente Um componente funcional simples em React Native:

```
1 import React from 'react';
2 import { View, Text, StyleSheet } from 'react-native';
3
4 const MeuPrimeiroComponente = () => {
5   return (
6     <View style={styles.container}>
7       <Text style={styles.titulo}>Olá, React Native!</Text>
8       <Text style={styles.subtitulo}>Este é o meu primeiro aplicativo móvel.</Text>
9     </View>
  
```

```

10 );
11 };
12
13 const styles = StyleSheet.create({
14   container: {
15     flex: 1,
16     justifyContent: 'center',
17     alignItems: 'center',
18     backgroundColor: '#F5FCFF',
19   },
20   titulo: {
21     fontSize: 24,
22     textAlign: 'center',
23     margin: 10,
24     color: '#333333',
25   },
26   subtitulo: {
27     fontSize: 18,
28     textAlign: 'center',
29     color: '#666666',
30   },
31 });
32
33 export default MeuPrimeiroComponente;

```

Navegação Para navegação entre telas, a biblioteca mais popular é o React Navigation.

- **Instalar React Navigation:**
- **Exemplo de Stack Navigator:**

Gerenciamento de Estado Assim como no React para web, o gerenciamento de estado é crucial em React Native. As opções incluem:

- **useState** e useContext (Hooks do React):**** Para estado local e global simples.
- **Redux, Zustand, MobX:** Para gerenciamento de estado mais complexo em aplicações maiores.

Requisições de Rede Para buscar dados de APIs, você pode usar a API `fetch` nativa do JavaScript ou bibliotecas como `axios`.

```

1 import React, { useEffect, useState } from 'react';
2 import { View, Text, FlatList, ActivityIndicator } from 'react-native';
3
4 const ListaDePosts = () => {
5   const [posts, setPosts] = useState([]);
6   const [loading, setLoading] = useState(true);
7
8   useEffect(() => {
9     fetch('https://jsonplaceholder.typicode.com/posts')
10      .then(response => response.json())
11      .then(data => {
12        setPosts(data);
13        setLoading(false);
14      })
15      .catch(error => {
16        console.error('Erro ao buscar posts:', error);
17        setLoading(false);
18      });
19   }, []);
20
21   if (loading) {
22     return (
23       <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
24         <ActivityIndicator size="large" color="#0000ff" />
25         <Text>Carregando posts...</Text>
26       </View>
27     );
28   }
29
30   return (
31     <View style={{ flex: 1, paddingTop: 20 }}>
32       <FlatList
33         data={posts}
34         keyExtractor={item => item.id.toString()}
35         renderItem={({ item }) => (
36           <View style={{ padding: 10, borderBottomWidth: 1, borderBottomColor: 'ccc' }}>
37             <Text style={{ fontSize: 18, fontWeight: 'bold' }}>{item.title}</Text>
38             <Text>{item.body}</Text>
39           </View>
40         )}
41       />
42     </View>
43   );
44 };
45
46 export default ListaDePosts;

```


Observações:

- React Native é uma ferramenta poderosa para o desenvolvimento móvel, mas exige uma compreensão dos conceitos de React e, para projetos mais avançados, familiaridade com o ambiente de desenvolvimento nativo.
- A comunidade React Native é muito ativa, com muitas bibliotecas e recursos disponíveis para estender as funcionalidades do seu aplicativo.

Documentação Oficial:

- [React Native: Documentação Oficial](#)
- [Expo: Documentação Oficial](#)
- [React Navigation: Documentação Oficial](#)

6.1. SQL Portátil: Comandos Essenciais

SQL (Structured Query Language) é a linguagem universal para interagir com bancos de dados relacionais. Embora existam dialetos específicos para cada sistema de gerenciamento de banco de dados (SGBD) como MySQL, PostgreSQL, SQL Server, etc., a maioria dos comandos básicos segue o padrão ANSI SQL, o que os torna portáteis entre diferentes sistemas. Esta seção foca nesses comandos universais.

Linguagem de Definição de Dados (DDL) DDL é usada para definir e gerenciar a estrutura do banco de dados e seus objetos, como tabelas.

Comando	Descrição	Exemplo de Uso	Observações
CREATE TABLE	Cria uma nova tabela no banco de dados.	CREATE TABLE usuarios (id INT PRIMARY KEY, nome VARCHAR(255) NOT NULL, email VARCHAR(255) UNIQUE);	Os tipos de dados (ex: INT, VARCHAR) são amplamente suportados, mas podem ter nomes ou limites diferentes em alguns SGBDs. AUTO_INCREMENT (MySQL) é SERIAL em PostgreSQL e IDENTITY em SQL Server.
ALTER TABLE	Modifica a estrutura de uma tabela existente.	ALTER TABLE usuarios ADD data_nascimento DATE; ALTER TABLE usuarios DROP COLUMN email;	A sintaxe para adicionar, remover ou modificar colunas é bastante padronizada.
DROP TABLE	Exclui permanentemente uma tabela e todos os seus dados.	DROP TABLE usuarios;	Ação irreversível. Use com cuidado.
CREATE INDEX	Cria um índice em uma ou mais colunas para acelerar as consultas.	CREATE INDEX idx_nome ON usuarios (nome);	Essencial para otimização de performance em tabelas grandes.

Linguagem de Manipulação de Dados (DML) DML é usada para gerenciar os dados dentro das tabelas.

Comando	Descrição	Exemplo de Uso
INSERT INTO	Adiciona uma ou mais novas linhas (registros) a uma tabela.	INSERT INTO usuarios (id, nome, email) VALUES (1, 'Ana Silva', 'ana.silva@example.com');

Comando	Descrição	Exemplo de Uso
UPDATE	Modifica registros existentes em uma tabela.	UPDATE usuarios SET email = 'ana.s@newdomain.com' WHERE id = 1;
DELETE	Remove registros de uma tabela.	DELETE FROM usuarios WHERE id = 1;

Linguagem de Consulta de Dados (DQL) DQL é usada para consultar e recuperar dados do banco de dados. O comando SELECT é o principal componente da DQL.

Cláusula/Operador	Descrição	Exemplo de Uso com SELECT
SELECT	Recupera dados de uma ou mais tabelas.	SELECT nome, email FROM usuarios; SELECT * FROM usuarios;
FROM	Especifica a tabela da qual os dados serão recuperados.	SELECT nome FROM usuarios;
WHERE	Filtra os registros com base em uma condição.	SELECT * FROM usuarios WHERE idade > 18;
ORDER BY	Ordena os resultados com base em uma ou mais colunas.	SELECT * FROM usuarios ORDER BY nome ASC; (ASC: ascendente, DESC: descendente)
GROUP BY	Agrupar linhas que têm os mesmos valores em colunas especificadas em linhas de resumo.	SELECT pais, COUNT(id) FROM usuarios GROUP BY pais;
HAVING	Filtra os resultados de um GROUP BY com base em uma condição.	SELECT pais, COUNT(id) FROM usuarios GROUP BY pais HAVING COUNT(id) > 10;

Funções Agregadas Padrão Funções que realizam um cálculo em um conjunto de valores e retornam um único valor.

Função	Descrição	Exemplo de Uso
COUNT()	Conta o número de linhas.	SELECT COUNT(*) FROM usuarios;
SUM()	Soma os valores de uma coluna numérica.	SELECT SUM(salario) FROM funcionarios;
AVG()	Calcula a média dos valores de uma coluna numérica.	SELECT AVG(idade) FROM usuarios;
MIN()	Retorna o menor valor de uma coluna.	SELECT MIN(preco) FROM produtos;
MAX()	Retorna o maior valor de uma coluna.	SELECT MAX(preco) FROM produtos;

Junções (JOINS) Combinam linhas de duas ou mais tabelas com base em uma coluna relacionada.

Tipo de JOIN	Descrição	Exemplo de Uso
INNER JOIN	Retorna registros que têm valores correspondentes em ambas as tabelas.	SELECT u.nome, p.nome_produto FROM usuarios u INNER JOIN pedidos p ON u.id = p.usuario_id;
LEFT JOIN	Retorna todos os registros da tabela da esquerda e os registros correspondentes da tabela da direita.	SELECT u.nome, p.nome_produto FROM usuarios u LEFT JOIN pedidos p ON u.id = p.usuario_id;
RIGHT JOIN	Retorna todos os registros da tabela da direita e os registros correspondentes da tabela da esquerda.	SELECT u.nome, p.nome_produto FROM usuarios u RIGHT JOIN pedidos p ON u.id = p.usuario_id;
FULL OUTER JOIN	Retorna todos os registros quando há uma correspondência na tabela da esquerda ou da direita.	SELECT u.nome, p.nome_produto FROM usuarios u FULL OUTER JOIN pedidos p ON u.id = p.usuario_id;

Observações:

- A sintaxe apresentada é amplamente compatível com a maioria dos SGBDs modernos. No

entanto, sempre consulte a documentação específica do seu SGBD para funcionalidades avançadas ou tipos de dados específicos.

Documentação de Referência:

- Padrões SQL (ANSI/ISO)
- W3Schools SQL Tutorial