Explorando o JavaScript

Ana Júlia Rita Lima Cardoso

Outubro de 2025

1 Arrays em JavaScript

1.1 O que são? Funcionam como listas do Python?

Um *array* em JavaScript é uma coleção ordenada de valores indexados a partir de zero. Assim como as `list` do Python, arrays em JS têm tamanho dinâmico. Diferenças importantes: podem misturar tipos, podem ser esparsos e possuem a propriedade `length` que, ao ser alterada, pode truncar ou expandir o array.

```
const misto = [1, "dois", { tres: 3 }, [4]];
console.log(misto[0]);     // 1
console.log(misto[2].tres); // 3
```

1.2 Como criar arrays em JS

Formas comuns de criação:

```
// Literal
const a = [10, 20, 30];

// Construtor
const b = new Array(3);    // [ , , ] (3 "buracos")
const c = Array(1, 2, 3);  // [1, 2, 3]

// A partir de iteráveis / array-like
const d = Array.from("abc"); // ['a', 'b', 'c']
const e = Array.of(5);       // [5]
```

1.3 Como acessar elementos

Use o índice ou o método `at` para índices negativos (ES2022+):

```
const arr = ["a", "b", "c"];
console.log(arr[0]);           // "a"
console.log(arr[arr.length-1]); // "c"
console.log(arr.at(-1));       // "c"
```

Desestruturação:

```
const [primeiro, segundo, ...resto] = [10, 20, 30, 40];
```

1.4 Propriedades e métodos fundamentais

`length` Retorna o tamanho do array e é mutável (pode truncar/expandir).

```
const v = [1, 2, 3, 4];
v.length = 2; // truncar
console.log(v); // [1, 2]
```

`push/pop` Adicionam/removem no final (mutáveis).

```
const pilha = [1, 2];
pilha.push(3);      // [1,2,3]
const x = pilha.pop(); // x=3
```

`unshift/shift` Adicionam/removem no início (mutáveis).

```
const fila = [2, 3];
fila.unshift(1); // [1,2,3]
const y = fila.shift(); // y=1
```

`indexOf` Retorna o índice da primeira ocorrência ou `-1`.

```
const nomes = ["Ana", "Bruno", "Ana"];
nomes.indexOf("Ana");     // 0
nomes.indexOf("Ana", 1);  // 2
```

`splice` Remove/insere em qualquer posição (muta e retorna removidos).

```
const nums = [1, 2, 3, 4];
const removidos = nums.splice(1, 2); // [2,3], nums=[1,4]
nums.splice(1, 0, 99); // inserir sem remover => [1,99,4]
```

sort Ordena *in-place*. Sem `compareFn`, a ordem é lexicográfica (string).

```
const a1 = [10, 2, 30];
a1.sort(); // ordem como strings, pode ser indesejada
a1.sort((x, y) => x - y); // numérico

const a2 = [10, 2, 30];
const ordenado = [...a2].sort((x, y) => x - y); // sem mutar a2
```

1.5 Formas de copiar e criar cópias

Cópias rasas (shallow) : copiam apenas o primeiro nível.

```
const original = [1, { nome: "Ana" }, [10, 20]];
const c1 = original.slice();
const c2 = [...original];
const c3 = Array.from(original);
c2[1].nome = "Bruno"; // afeta original[1]
```

Cópias profundas (deep) : necessárias quando há aninhamento.

```
// Profunda com API moderna:
const deep = structuredClone(original);

// Copiando uma matriz 2D:
const matriz = [[1,2],[3,4]];
const copiaMatriz = matriz.map(linha => linha.slice());
```

1.6 Arrays multidimensionais

Criação e acesso

```
const grid = [
  [1, 2, 3],
  [4, 5, 6]
];
console.log(grid[0][1]); // 2
```

Criando uma matriz NxM corretamente

```
// ERRADO: linhas referenciam o mesmo array
const errado = Array(3).fill(Array(4).fill(0));
```

```
// CERTO: cada linha independente
const linhas = 3, colunas = 4;
const matrizOk = Array.from({ length: linhas }, () =>
  Array(colunas).fill(0)
);
```

Iteração

```
for (const linha of matrizOk) {
  for (const valor of linha) {
    // usar valor
  }
}
```

2 Estruturas de Condição em JavaScript

2.1 Pra que servem?

As estruturas de condição controlam o fluxo do programa, permitindo que ele execute blocos diferentes conforme uma expressão seja verdadeira (`true`) ou falsa (`false`).

2.2 `if`, `else if` e `else`

```
let idade = 18;

if (idade < 18) {
  console.log("Menor de idade");
} else if (idade === 18) {
  console.log("Tem exatamente 18 anos");
} else {
  console.log("Maior de idade");
}
```

O bloco entre chaves é executado somente se a condição for verdadeira. Valores *falsy* (como `0`, , `null`, `undefined`, `NaN`) são interpretados como falso.

```
if ("") {
  console.log("não executa");
}
if ("texto") {
  console.log("executa"); // string não vazia é true
}
```

2.3 `switch` e `case`

```
let dia = 3;
let nomeDia;

switch (dia) {
  case 1:
    nomeDia = "Domingo";
    break;
  case 2:
    nomeDia = "Segunda";
    break;
  case 3:
    nomeDia = "Terça";
    break;
  default:
    nomeDia = "Dia inválido";
}

console.log(nomeDia); // "Terça"
```

O `switch` compara uma variável com vários valores possíveis. O `break` evita a execução em cascata. O `default` é opcional e equivale ao "else".

2.4 Diferenças entre `if/else` e `switch`

Aspecto	**if / else**	**switch / case**
Uso	Comparações lógicas complexas	Igualdade simples
Tipo de expressão	Qualquer expressão booleana	Geralmente um único valor
Clareza	Melhor para poucas condições	Melhor para muitos casos fixos
Necessidade de break	Não	Sim

2.5 Operadores de Comparação

Relacionais

Operador	Significado	Exemplo	Resultado
$>$	Maior que	$5 > 3$	true
$<$	Menor que	$2 < 1$	false
$>=$	Maior ou igual	$10 >= 10$	true
$<=$	Menor ou igual	$7 <= 8$	true

Igualdade e Identidade

Operador	Significado	Exemplo	Resultado
==	Igual (com coerção)	”5”== 5	true
!=	Diferente (com coerção)	”5”!= 5	false
===	Igual (sem coerção)	”5”=== 5	false
!==	Diferente (sem coerção)	”5”!== 5	true

2.6 Operadores Lógicos

```
let idade = 20;
let temCarteira = true;

if (idade >= 18 && temCarteira) {
  console.log("Pode dirigir");
}

if (idade < 18 || !temCarteira) {
  console.log("Não pode dirigir");
}
```

Operador	Nome	Descrição	Exemplo
&&	E lógico	Verdadeiro se ambos forem verdadeiros	true && false = false
||	OU lógico	Verdadeiro se pelo menos um for verdadeiro	true || false = true
!	NÃO lógico	Inverte o valor lógico	!true = false

2.7 Operadores Aritméticos

```
let x = 5;
x++;
console.log(x); // 6

x *= 2; // mesmo que x = x * 2
console.log(x); // 12
```

Operador	Descrição	Exemplo	Resultado
+	Soma	5 + 3	8
-	Subtração	5 - 3	2
*	Multiplicação	4 * 2	8
/	Divisão	8 / 2	4
%	Resto da divisão	7 % 3	1
**	Exponenciação	2 ** 3	8
++	Incremento	x++	Soma 1
–	Decremento	x–	Subtrai 1

3 Estruturas de Repetição em JavaScript

3.1 Pra que servem?

As estruturas de repetição permitem executar um bloco de código várias vezes enquanto uma condição for verdadeira. São usadas para automatizar tarefas repetitivas, percorrer coleções e controlar loops de interação.

3.2 `while`

Executa enquanto a condição for verdadeira. A condição é avaliada **antes** de cada repetição.

```
let contador = 0;

while (contador < 5) {
  console.log("Contador:", contador);
  contador++;
}
```

Se a condição inicial for falsa, o bloco não é executado nenhuma vez.

3.3 `do...while`

Executa o bloco ao menos uma vez e só depois verifica a condição.

```
let senha;

do {
  senha = prompt("Digite sua senha:");
} while (senha !== "1234");

console.log("Acesso permitido!");
```

Mesmo com a condição falsa desde o início, o corpo do laço é executado ao menos uma vez.

3.4 `for`

Usado quando o número de repetições é conhecido.

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

O `for` possui três partes:

- Inicialização: `let i = 0`
- Condição: `i < 5`
- Incremento: `i++`

3.4.1 Percorrendo arrays

```
const frutas = ["maçã", "banana", "uva"];
for (let i = 0; i < frutas.length; i++) {
  console.log(frutas[i]);
}
```

3.4.2 `for...of` — percorre valores

```
for (const fruta of frutas) {
  console.log(fruta);
}
```

3.4.3 `for...in` — percorre índices ou chaves

```
for (const i in frutas) {
  console.log(i, frutas[i]);
}
```

3.5 `break` e `continue`

`break` interrompe o laço completamente. `continue` pula para a próxima iteração.

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue; // pula o 3
  if (i === 5) break;    // encerra o laço
  console.log(i);
}
// Saída: 1, 2, 4
```

3.6 Diferenças entre os tipos de laço

Estrutura	Quando usar	Avaliação da condição	Executa pelo menos 1 v
`while`	Condição desconhecida	Antes do bloco	Não
`do...while`	Deve rodar ao menos uma vez	Depois do bloco	Sim
`for`	Contagem conhecida	Antes de cada iteração	Não
`for...of`	Percorrer valores	Implícita	Depende do iterável
`for...in`	Percorrer índices/chaves	Implícita	Depende do objeto