Arrays

Arrays são estruturas de dados que permitem armazenar uma coleção ordenada de valores. Pense neles como uma lista ou uma caixa com várias divisórias, onde cada divisória pode guardar um item diferente. Em JavaScript, um array pode conter valores de diferentes tipos de dados (números, strings, booleanos, objetos, etc.), embora seja uma boa prática manter o mesmo tipo de dado para facilitar a manipulação.

2.1. Declaração e Inicialização

Você pode declarar e inicializar um array de várias maneiras. A forma mais comum é usando colchetes [] .

Exemplo 2.1.1: Declaração de um array vazio

```
let meuArrayVazio = [];
console.log(meuArrayVazio); // Saída: []
```

Exemplo 2.1.2: Declaração e inicialização com valores

```
let frutas = ["Maçã", "Banana", "Laranja"];
console.log(frutas); // Saída: [ 'Maçã', 'Banana', 'Laranja' ]

let numeros = [1, 2, 3, 4, 5];
console.log(numeros); // Saída: [ 1, 2, 3, 4, 5 ]

let misto = ["Texto", 10, true, null];
console.log(misto); // Saída: [ 'Texto', 10, true, null ]
```

2.2. Acessando Elementos (Índices)

Os elementos em um array são acessados por meio de um índice numérico, que começa em `0` para o primeiro elemento. Isso significa que se um array tem N elementos, os índices vão de `0` a `N-1` .

Exemplo 2.2.1: Acessando elementos específicos

```
let cores = ["Vermelho", "Verde", "Azul"];
console.log(cores[0]); // Saída: Vermelho
console.log(cores[1]); // Saída: Verde
console.log(cores[2]); // Saída: Azul
console.log(cores[3]); // Saída: undefined (índice fora do limite)
```

Você também pode modificar um elemento existente usando seu índice:

Exemplo 2.2.2: Modificando elementos

```
let carros = ["Fiat", "Ford", "Chevrolet"];
console.log(carros); // Saída: [ 'Fiat', 'Ford', 'Chevrolet' ]

carros[0] = "Volkswagen";
console.log(carros); // Saída: [ 'Volkswagen', 'Ford', 'Chevrolet' ]
```

2.3. Propriedade `length`

A propriedade `length` retorna o número de elementos em um array. É muito útil para saber o tamanho do seu array ou para iterar sobre ele.

Exemplo 2.3.1: Usando a propriedade length

```
let paises = ["Brasil", "Argentina", "Chile", "Colômbia"];
console.log(paises.length); // Saída: 4

paises.push("Peru"); // Adiciona um elemento
console.log(paises.length); // Saída: 5
```

2.4. Métodos Comuns de Arrays

JavaScript oferece uma rica coleção de métodos para manipular arrays. Vamos explorar os mais comuns e úteis.

2.4.1. `push()` e `pop()`

`push()` : Adiciona um ou mais elementos ao final do array e retorna o novo comprimento do array.

pop() : Remove o último elemento de um array e retorna esse elemento.

Exemplo 2.4.1.1: Usando push() e pop()

```
 let animais = ["Cachorro", "Gato", "Pássaro"];
console.log(animais); // Saída: [ 'Cachorro', 'Gato', 'Pássaro' ]

animales.push("Peixe");
console.log(animais); // Saída: [ 'Cachorro', 'Gato', 'Pássaro', 'Peixe' ]

let ultimoAnimal = animais.pop();
console.log(ultimoAnimal); // Saída: Peixe
console.log(animais); // Saída: [ 'Cachorro', 'Gato', 'Pássaro' ]
```

2.4.2. shift() e unshift()

shift() : Remove o primeiro elemento de um array e retorna esse elemento. Todos os outros elementos são deslocados para uma posição inferior.

unshift() : Adiciona um ou mais elementos ao início do array e retorna o novo comprimento do array.

Exemplo 2.4.2.1: Usando shift() e unshift()

```
 let fila = ["João", "Maria", "Pedro"];
console.log(fila); // Saída: [ 'João', 'Maria', 'Pedro' ]

let primeiroDaFila = fila.shift();
console.log(primeiroDaFila); // Saída: João
console.log(fila); // Saída: [ 'Maria', 'Pedro' ]

fila.unshift("Ana", "Carlos");
console.log(fila); // Saída: [ 'Ana', 'Carlos', 'Maria', 'Pedro' ]
```

2.4.3. splice()

O método splice() é um dos mais versáteis para arrays. Ele pode adicionar, remover ou substituir elementos em qualquer posição do array.

Sintaxe: array.splice(start, deleteCount, item1, item2, ...)

`start` : Índice no qual a modificação deve começar.

`deleteCount` : Número de elementos a serem removidos a partir do `start` . Se for `0` , nenhum elemento é removido.

`item1, item2, ...` : Elementos a serem adicionados ao array a partir do `start` .

Exemplo 2.4.3.1: Removendo elementos com splice()

```
 let nomes = ["Alice", "Bob", "Charlie", "David"];
console.log(nomes); // Saída: [ 'Alice', 'Bob', 'Charlie', 'David' ]

// Remover 1 elemento a partir do índice 2 (Charlie)
let removidos = nomes.splice(2, 1);
console.log(removidos); // Saída: [ 'Charlie' ]
console.log(nomes); // Saída: [ 'Alice', 'Bob', 'David' ]
```

Exemplo 2.4.3.2: Adicionando elementos com splice()

```
 let cidades = ["São Paulo", "Rio de Janeiro", "Belo Horizonte"];
console.log(cidades); // Saída: [ 'São Paulo', 'Rio de Janeiro', 'Belo
Horizonte' ]

// Adicionar "Curitiba" e "Porto Alegre" a partir do índice 1, sem remover nada
cidades.splice(1, 0, "Curitiba", "Porto Alegre");
console.log(cidades); // Saída: [ 'São Paulo', 'Curitiba', 'Porto Alegre', 'Rio de Janeiro',
'Belo Horizonte' ]
```

Exemplo 2.4.3.3: Substituindo elementos com splice()

```
 let produtos = ["TV", "Geladeira", "Fogão"];
console.log(produtos); // Saída: [ 'TV', 'Geladeira', 'Fogão' ]

// Substituir "Geladeira" por "Máquina de Lavar"
produtos.splice(1, 1, "Máquina de Lavar");
console.log(produtos); // Saída: [ 'TV', 'Máquina de Lavar', 'Fogão' ]
```

2.4.4. slice()

O método `slice()` retorna uma cópia rasa de uma parte de um array em um novo array. Ele não modifica o array original.

Sintaxe: `array.slice(start, end)`

`start` : Índice onde a extração deve começar (inclusive). Se omitido, começa do índice 0.

`end` : Índice onde a extração deve terminar (exclusive). Se omitido, extrai até o final do array.

Exemplo 2.4.4.1: Usando slice()

```
let letras = ["a", "b", "c", "d", "e", "f"];
console.log(letras); // Saída: [ 'a', 'b', 'c', 'd', 'e', 'f' ]

let subArray1 = letras.slice(2, 5); // Extrai do índice 2 (inclusive) ao 5 (exclusive)
console.log(subArray1); // Saída: [ 'c', 'd', 'e' ]

let subArray2 = letras.slice(1); // Extrai do índice 1 até o final
console.log(subArray2); // Saída: [ 'b', 'c', 'd', 'e', 'f' ]

let copiaCompleta = letras.slice(); // Cria uma cópia completa do array
console.log(copiaCompleta); // Saída: [ 'a', 'b', 'c', 'd', 'e', 'f' ]
```

2.4.5. indexOf() e includes()

`indexOf()` : Retorna o primeiro índice em que um determinado elemento pode ser encontrado no array, ou `-1` se não estiver presente.

`includes()` : Determina se um array inclui um determinado valor, retornando `true` ou `false` .

Exemplo 2.4.5.1: Usando indexOf() e includes()

```
let frutasDisponiveis = ["Maçã", "Banana", "Laranja", "Maçã"];
console.log(frutasDisponiveis.indexOf("Banana")); // Saída: 1
console.log(frutasDisponiveis.indexOf("Maçã")); // Saída: 0 (retorna o
primeiro)
console.log(frutasDisponiveis.indexOf("Uva")); // Saída: -1

console.log(frutasDisponiveis.includes("Laranja")); // Saída: true
console.log(frutasDisponiveis.includes("Pera")); // Saída: false
```

2.5. Iteração com Arrays

Iterar sobre arrays é uma tarefa comum. Vamos ver as duas formas mais utilizadas: o loop `for` tradicional e o método `forEach()` .

2.5.1. Loop for

O loop `for` é uma estrutura de controle que permite executar um bloco de código repetidamente. É ideal para iterar sobre arrays quando você precisa de controle sobre o

índice ou quando precisa parar a iteração em um determinado ponto.

Exemplo 2.5.1.1: Iterando com loop for

```
let alunos = ["Carlos", "Fernanda", "Gustavo", "Helena"];

for (let i = 0; i < alunos.length; i++) {
    console.log(`Aluno no índice ${i}: ${alunos[i]}`);
}
/* Saída:
Aluno no índice 0: Carlos
Aluno no índice 1: Fernanda
Aluno no índice 2: Gustavo
Aluno no índice 3: Helena
 */
```

2.5.2. Método forEach()

O método forEach() executa uma função fornecida uma vez para cada elemento do array. É uma forma mais concisa e legível de iterar sobre arrays quando você não precisa do índice ou não precisa parar a iteração.

Exemplo 2.5.2.1: Iterando com forEach()

```
let times = ["Flamengo", "Palmeiras", "Grêmio"];

times.forEach(function(time, indice) {
    console.log(`Time no índice ${indice}: ${time}`);
});
/* Saída:
Time no índice 0: Flamengo
Time no índice 1: Palmeiras
Time no índice 2: Grêmio
 */

// Com arrow function (sintaxe mais moderna)
times.forEach((time) => {
    console.log(`Time: ${time}`);
});
/* Saída:
Time: Flamengo
Time: Palmeiras
Time: Grêmio
 */
```

3. Operadores

Operadores são símbolos especiais que realizam operações em um ou mais valores (operandos) e produzem um resultado. Eles são a espinha dorsal de qualquer linguagem de programação, permitindo que você manipule dados, tome decisões e controle o fluxo do seu programa. Em JavaScript, temos diversos tipos de operadores, cada um com sua finalidade específica.

3.1. Tipos de Operadores

3.1.1. Operadores Aritméticos

Usados para realizar cálculos matemáticos.

Operador	Descrição	Exemplo	Resultado
+	Adição	5 + 3	8
-	Subtração	10 - 4	6
*	Multiplicação	2 * 6	12
/	Divisão	15 / 3	5
%	Módulo (resto)	10 % 3	1
**	Exponenciação	2 ** 3	8

Exemplo 3.1.1.1: Operadores Aritméticos

```
 let a = 10;
let b = 3;

console.log(a + b); // Saída: 13
console.log(a - b); // Saída: 7
console.log(a * b); // Saída: 30
console.log(a / b); // Saída: 3.3333333333333335
console.log(a % b); // Saída: 1
console.log(a ** b); // Saída: 1000
```

3.1.2. Operadores de Atribuição

Usados para atribuir valores a variáveis. O operador de atribuição básico é o `=` . Existem também operadores de atribuição compostos que combinam uma operação aritmética com uma atribuição.

Operador	Exemplo	Equivalente a
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2
%=	x %= 3	x = x % 3
**=	x **= 2	x = x ** 2

Exemplo 3.1.2.1: Operadores de Atribuição

```
 let x = 10;
console.log(x); // Saída: 10

x += 5; // x agora é 15
console.log(x); // Saída: 15

x -= 3; // x agora é 12
console.log(x); // Saída: 12

x *= 2; // x agora é 24
console.log(x); // Saída: 24

x /= 4; // x agora é 6
console.log(x); // Saída: 6

x %= 5; // x agora é 1
```

```
console.log(x); // Saída: 1

x = 2;
x **= 3; // x agora é 8
console.log(x); // Saída: 8
```

3.1.3. Operadores de Comparação

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

Operador	Descrição	Exemplo	Resultado
`==`	Igual a (compara valor, ignora tipo)	`5 == "5"`	`true`
`===`	Estritamente igual a (compara valor e tipo)	`5 === "5"`	`false`
`!=`	Diferente de (compara valor, ignora tipo)	`10 != "10"`	`false`
`!==`	Estritamente diferente de (compara valor e tipo)	`10 !== "10"`	`true`
`>`	Maior que	`7 > 5`	`true`
`<`	Menor que	`3 < 8`	`true`
`>=`	Maior ou igual a	`5 >= 5`	`true`
`<=`	Menor ou igual a	`4 <= 6`	`true`

Exemplo 3.1.3.1: Operadores de Comparação

```
 let num1 = 10;
let num2 = "10";
let num3 = 20;

console.log(num1 == num2); // Saída: true (compara apenas o valor)
console.log(num1 === num2); // Saída: false (compara valor e tipo)
console.log(num1 != num3); // Saída: true
console.log(num1 !== num2); // Saída: true
```

```
console.log(num3 > num1); // Saída: true
console.log(num1 < num3); // Saída: true
console.log(num1 >= 10); // Saída: true
console.log(num3 <= 10); // Saída: false
```

3.1.4. Operadores Lógicos

Usados para combinar ou inverter valores booleanos.

Operador	Descrição	Exemplo	Resultado
&&	AND lógico	true && false	false
||	OR lógico	true || false	true
!	NOT lógico	!true	false

Exemplo 3.1.4.1: Operadores Lógicos

```
let idade = 18;
let temCnh = true;

console.log(idade >= 18 && temCnh); // Saída: true (ambas as condições são verdadeiras)

let temDinheiro = false;
let temCartao = true;

console.log(temDinheiro || temCartao); // Saída: true (pelo menos uma condição é verdadeira)

let estaChovendo = false;
console.log(!estaChovendo); // Saída: true (inverte o valor booleano)
```

3.1.5. Operadores Unários

Operadores que atuam sobre um único operando.

Operador	Descrição	Exemplo	Resultado
`++`	Incremento (adiciona 1)	`x++` ou `++x`	
`--`	Decremento (subtrai 1)	`x--` ou `--x`	

Exemplo 3.1.5.1: Operadores Unários (Incremento/Decremento)

```
 let contador = 5;
console.log(contador); // Saída: 5

contador++; // Equivalente a contador = contador + 1
console.log(contador); // Saída: 6

contador--; // Equivalente a contador = contador - 1
console.log(contador); // Saída: 5

let preIncremento = ++contador; // Incrementa e depois atribui
console.log(preIncremento); // Saída: 6
console.log(contador); // Saída: 6

let posIncremento = contador++; // Atribui e depois incrementa
console.log(posIncremento); // Saída: 6
console.log(contador); // Saída: 7
```

3.1.6. Operador Ternário (Condicional)

É um atalho para a instrução `if...else` . Ele avalia uma condição e retorna um valor se a condição for verdadeira e outro valor se for falsa.
Sintaxe: `condição ? valor_se_verdadeiro : valor_se_falso`

Exemplo 3.1.6.1: Operador Ternário

```
 let idadeUsuario = 20;
let status = (idadeUsuario >= 18) ? "Maior de idade" : "Menor de idade";
console.log(status); // Saída: Maior de idade

let estaComFome = true;
let acao = estaComFome ? "Comer" : "Esperar";
console.log(acao); // Saída: Comer
```

3.2. Precedência de Operadores

A precedência de operadores determina a ordem em que os operadores são avaliados em uma expressão. Operadores com maior precedência são avaliados primeiro. Quando operadores têm a mesma precedência, a associatividade (da esquerda para a direita ou da direita para a esquerda) determina a ordem.

Por exemplo, a multiplicação e a divisão têm maior precedência que a adição e a subtração.

Exemplo 3.2.1: Precedência de Operadores

```
let resultado = 5 + 3 * 2; // Multiplicação (3 * 2 = 6) é feita antes da adição (5 + 6 = 11)
console.log(resultado); // Saída: 11

let resultadoComParenteses = (5 + 3) * 2; // Parênteses alteram a precedência (8 * 2 = 16)
console.log(resultadoComParenteses); // Saída: 16
```

É sempre uma boa prática usar parênteses `()` para garantir a ordem de avaliação desejada e para tornar seu código mais legível, mesmo que a precedência padrão já garanta o resultado correto.

4. Expressões

Em programação, uma **expressão** é uma combinação de valores, variáveis, operadores e chamadas de função que o JavaScript interpreta para produzir um único valor. Toda expressão resulta em um valor. Compreender as expressões é fundamental para escrever código que realiza cálculos, comparações e manipula dados de forma eficaz.

4.1. O que são Expressões?

Basicamente, qualquer coisa que produz um valor é uma expressão. Isso contrasta com as **declarações (statements)**, que realizam uma ação, mas não necessariamente produzem um valor (como `if` , `for` , `while`).

Exemplo 4.1.1: Exemplos simples de expressões

```
5; // Uma expressão literal, o valor é 5
"Olá Mundo"; // Uma expressão literal, o valor é "Olá Mundo"
let x = 10; // A atribuição é uma declaração, mas `10` é uma expressão x; //
Uma expressão que avalia o valor de x
```

4.2. Tipos de Expressões

Vamos categorizar as expressões com base nos operadores que as compõem.

4.2.1. Expressões Aritméticas

Combinam valores numéricos e operadores aritméticos para produzir um novo valor numérico.

Exemplo 4.2.1.1: Expressões Aritméticas

```
let preco = 25;
let quantidade = 3;
let total = preco * quantidade; // Expressão aritmética: 25 * 3 resulta em 75
console.log(total); // Saída: 75

let media = (8 + 7 + 9) / 3; // Expressão aritmética complexa
console.log(media); // Saída: 8
```

4.2.2. Expressões de Comparação

Utilizam operadores de comparação para comparar dois valores e resultam em um valor booleano (`true` ou `false`).

Exemplo 4.2.2.1: Expressões de Comparação

```
let idadeMinima = 18;
let idadeAtual = 20;
let podeDirigir = idadeAtual >= idadeMinima; // Expressão de comparação: 20 >= 18
resulta em true
console.log(podeDirigir); // Saída: true

let nome1 = "Ana";
let nome2 = "ana";
let nomesIguais = nome1 === nome2; // Expressão de comparação: "Ana" === "ana"
resulta em false
console.log(nomesIguais); // Saída: false
```

4.2.3. Expressões Lógicas

Combinam expressões booleanas usando operadores lógicos (`&&` , || , `!`) para produzir um único valor booleano.

Exemplo 4.2.3.1: Expressões Lógicas

```
 let usuarioLogado = true;
let temPermissao = false;
let podeAcessar = usuarioLogado && temPermissao; // Expressão lógica: true && false resulta em false
console.log(podeAcessar); // Saída: false

let temCafe = true;
let temLeite = false;
let podeFazerCafeComLeite = temCafe || temLeite; // Expressão lógica: true || false resulta em true
console.log(podeFazerCafeComLeite); // Saída: true

let estaAtivo = false;
let estaInativo = !estaAtivo; // Expressão lógica: !false resulta em true
console.log(estaInativo); // Saída: true
```

4.2.4. Expressões de Atribuição

Embora a atribuição (`=`) seja uma declaração, a expressão do lado direito da atribuição é avaliada para produzir um valor que será atribuído à variável. Além disso, a própria operação de atribuição em JavaScript também retorna o valor que foi atribuído.

Exemplo 4.2.4.1: Expressões de Atribuição

```
 let valor = 10 + 5; // A expressão `10 + 5` é avaliada para 15, que é então atribuído a `valor`
console.log(valor); // Saída: 15

let a = 5;
let b = (a = 10); // A expressão `a = 10` atribui 10 a `a` e retorna 10, que é então atribuído a `b`
console.log(a); // Saída: 10
console.log(b); // Saída: 10
```

4.3. Expressões em Contexto

Expressões são usadas em quase todas as partes do seu código JavaScript. Elas são a base para:

Cálculos: `area = largura * altura;`

Condições: `if (idade >= 18) { ... }`

Loops: `for (let i = 0; i < array.length; i++) { ... }`

Retorno de funções: `return a + b;`

Argumentos de funções: `console.log("O resultado é: " + (x * y));`

Compreender como as expressões são avaliadas e o valor que elas produzem é crucial para escrever código JavaScript eficaz e sem erros.