

Javascript

Arrays

○ *Array*

- ▷ Podemos armazenar vários elementos em objetos, mas e se quiséssemos armazenar elementos em uma ordem?
 - Para esse caso, é melhor é utilizar um *array*.
- ▷ *Arrays* são conhecidos como vetores quando unidimensionais e como matrizes quando possuem mais de uma dimensão.
- ▷ Veja a sintaxe para criação de um *array*:

```
const valores = [];
```

Atribuindo valores na declaração

- ▷ É possível atribuir valores em um *array* em sua declaração:

```
const valores = [1, 5, 10, 27];
```

```
const nomes = ['Ana', 'Bruno', 'Carlos'];
```

Conhecendo o tamanho do *array*

- ▷ O tamanho do *array* pode ser obtido através da propriedade `length`:

```
console.log(valores.length);  
console.log(nomes.length);
```

- ▷ Será impresso:

4

3

Obtendo elementos pelo índice

- ▶ Você pode acessar um elemento pelo seu índice. Veja o vetor nomes:

```
▼ (3) ['Ana', 'Bruno', 'Carlos']  
  0: "Ana"  
  1: "Bruno"  
  2: "Carlos"  
length: 3
```

```
console.log('Elemento na posição 0: ', nomes[0]);  
console.log('Elemento na posição 1: ', nomes[1]);  
console.log('Elemento na posição 2: ', nomes[2]);
```

Obtendo elementos pelo índice

- ▷ Resultado impresso:

Elemento na posição 0: Ana

Elemento na posição 1: Bruno

Elemento na posição 2: Carlos

▼ (3) ['Ana', 'Bruno', 'Carlos']
0: "Ana"
1: "Bruno"
2: "Carlos"
length: 3

Alterando elementos

- ▷ Alterando um elemento:

```
nomes[0] = 'Ana Maria';  
nomes[1] = 'Bruno da Silva';
```

- ▷ Resultado:


```
0: "Ana Maria"  
1: "Bruno da Silva"  
2: "Carlos"  
length: 3
```

Adicionando elementos no final

- ▶ Para **adicionar** um elemento ao **final** do vetor, utilize o método **push**

```
const valores = [1, 5, 10, 27];  
valores.push(30);  
console.log(valores);
```

▼ (4) [1, 5, 10, 27] ⓘ
0: 1
1: 5
2: 10
3: 27
4: 30
length: 5




Adicionando elementos no final

- ▷ É possível adicionar mais de um elemento no final

```
const valores = [1, 5, 10, 27];  
valores.push(40, 45);  
console.log(valores);
```

▼ (6) [1, 5, 10, 27, 40, 45] ⓘ


0:	1
1:	5
2:	10
3:	27
4:	40
5:	45
length:	6



Adicionando elementos no início

- ▶ Para **adicionar** um elemento no **início** do vetor, utilize o método **unshift**

```
const valores = [1, 5, 10, 27];  
valores.unshift(12);  
console.log(valores);
```



▼ (5) [12, 1, 5, 10, 27] ⓘ
0: 12
1: 1
2: 5
3: 10
4: 27
length: 5

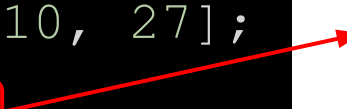
Adicionando elementos no início

- ▷ É possível adicionar mais de um elemento no início

```
const valores = [1, 5, 10, 27];  
valores.unshift(-5, 0);  
console.log(valores);
```

▼ (6) [-5, 0, 1, 5, 10, 27] ⓘ

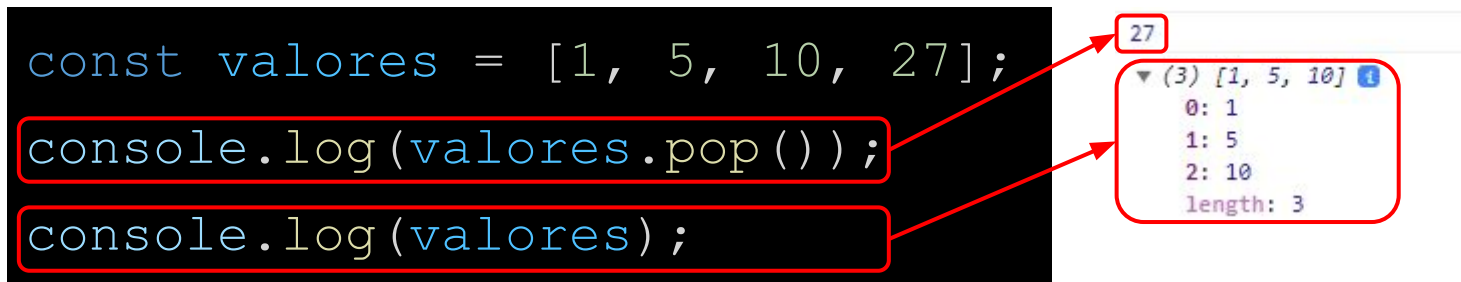
0:	-5
1:	0
2:	1
3:	5
4:	10
5:	27
length:	6



Retornando e removendo elementos

- ▷ Retornando e removendo o último elemento do vetor:

```
const valores = [1, 5, 10, 27];  
console.log(valores.pop());  
console.log(valores);
```



27

(3) [1, 5, 10]

0: 1

1: 5

2: 10

length: 3

Retornando e removendo elementos

- ▷ Retornando e removendo o primeiro elemento do vetor:

```
const valores = [1, 5, 10, 27];  
console.log(valores.shift());  
console.log(valores);
```

1

▼ (3) [5, 10, 27]

0: 5
1: 10
2: 27
length: 3

Percorrendo arrays

for, forEach

Percorrendo *arrays*

- ▷ Você pode percorrer um array utilizando uma estrutura de repetição para variar os seus índices

```
const valores = [1, 5, 10];
```

```
for (i = 0; i < valores.length; i++) {  
    console.log(valores[i] * 2);  
}
```



2

10

20

Percorrendo *arrays*

- ▷ Você pode percorrer um array utilizando uma estrutura de repetição para variar os seus índices também assim

```
const valores = [1, 5, 10];
```

```
for (let i in valores) {  
  console.log(valores[i] * 3);  
}
```

3

15

30

forEach

- ▷ O método `forEach` permite a iteração para cada elemento de um *array*.
- ▷ Este método recebe uma função e, para cada elemento no *array*, chama o método recebido com os seguintes parâmetros:
 - `item`
 - Valor do elemento
 - `index`
 - Índice do elemento no *array*
 - `array`
 - O próprio *array*

forEach

▷ Exemplo:

```
[1,2,5].forEach((item, index, array) =>
  console.log(item, index, array));
```

1 0 ▶ (3) [1, 2, 5]

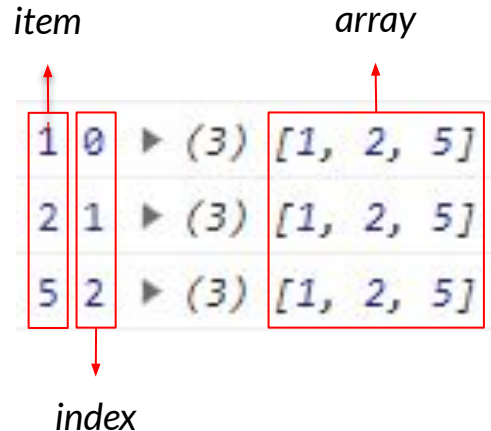
2 1 ▶ (3) [1, 2, 5]

5 2 ▶ (3) [1, 2, 5]

forEach

▷ Exemplo:

```
[1,2,5].forEach((item, index, array) =>
  console.log(item, index, array));
```



forEach

- ▷ Não é necessário receber todos os parâmetros:

```
[1,2,5].forEach(x => console.log(x));
```

1
2
5

forEach

- ▷ Outro exemplo do uso do forEach:

```
let soma = 0;  
[1,2,5].forEach(x => soma += x);  
console.log('Soma:', soma);
```

Soma: 8

Busca em *arrays*

indexOf, lastIndexOf, includes

Diferenças entre indexOf e includes

- ▷ Os métodos indexOf, lastIndexOf e includes pesquisam os itens à partir de um índice.
- ▷ Existe uma diferença no funcionamento destes métodos:
 - os métodos indexOf e lastIndexOf retornam -1 caso não encontrem ou retornam o índice do elemento encontrado.
 - o método include retorna false caso não encontre ou retorna true caso encontre o elemento no *array*.

indexOf e lastIndexOf

- ▷ O método `indexOf` busca os elementos do array a partir do início do mesmo.
- ▷ O método `lastIndexOf` busca os elementos do array a partir do final do mesmo, seguindo do maior índice para o menor índice.
- ▷ Assim que o elemento é encontrado, estes métodos retornam o índice.
- ▷ Se não encontrado, retornam -1

indexOf e lastIndexOf

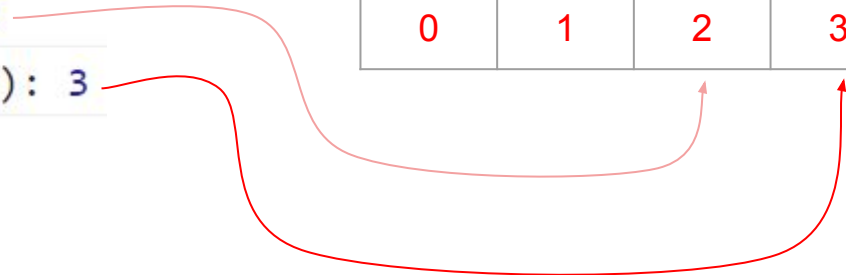


```
const elementos = [1,2,5,5,4,3];  
console.log('indexOf(5):', elementos.indexOf(5));  
console.log('lastIndexOf(5):', elementos.lastIndexOf(5));
```

indexOf(5): 2

lastIndexOf(5): 3

1	2	5	5	4	3
0	1	2	3	4	5



Transformando *arrays*

map, sort, reverse

map

- ▷ O método `map` chama uma função para cada elemento do *array* e retorna o *array* dos resultados da chamada desta função.

```
const elementos = [1,2,5,5,4,3];
```

```
const dobroElementos = elementos.map(x => x * 2);
```

```
console.log(elementos);
```

```
console.log(dobroElementos);
```

▶ (6) [1, 2, 5, 5, 4, 3]

▶ (6) [2, 4, 10, 10, 8, 6]


sort

- ▷ O método `sort` ordena os elementos do *array*. Este método também retorna o *array*, porém é comum não aproveitar o retorno desta função, uma vez que ela altera o próprio *array* ordenado.

```
const elementos = [1,2,15,4,3];
```

```
elementos.sort();
```

```
console.log(elementos);
```



► (5) [1, 15, 2, 3, 4]

Todos os elementos são ordenados como sendo strings por padrão. Por este motivo o 15 está antes do 2.

sort

- ▷ É possível fornecer uma função de comparação:

```
function compare(a, b) {  
    if (a > b) return 1;  
    if (a == b) return 0;  
    return -1; //if (a < b)  
}
```

```
const elementos = [1,2,15,4,3];
```

```
elementos.sort(compare);
```

```
console.log(elementos);
```

▶ (5) [1, 2, 3, 4, 15]

sort


- ▷ É aconselhável utilizar a função `localeCompare` para ordenar strings:

```
const palavras = ["Água", "Amendoa", "Batata"];  
palavras.sort();  
console.log(palavras);
```



```
▶ (3) ['Amendoa', 'Batata', 'Água']
```

```
const palavras = ["Água", "Amendoa", "Batata"];  
elementos.sort((a,b) => a.localeCompare(b));  
console.log(palavras);
```




```
▶ (3) ['Água', 'Amendoa', 'Batata']
```

reverse

- ▷ O método `reverse` inverte a ordem dos elementos do *array*.

```
const elementos = [1,2,5,4];  
elementos.reverse();  
console.log(elementos);
```



► (4) [4, 5, 2, 1]

Fim de material

Dúvidas?