🎖️

# Arrays no JavaScript e seus métodos

> Arrays são estruturas de dados que contem valores organizados de forma indexada, sendo que o primeiro valor de um array ocupa a posição (index) 0, o segundo a posição (index) 1, e assim por diante, até a posição **n**, sendo essa, o último elemento do array. Os arrays são muito úteis em diversas situações de desenvolvimento, para organizar dados de maneira ordenada e mais organizada do que deixar tudo solto, os arrays são extremamente importantes na programação, seja em qual linguagem for, pois eles funcionam como uma caixa onde guardamos coisas dentro, porém uma caixa organizada, preste bastante atenção e treine muito principalmente a parte de manipular os arrays com os métodos que veremos, isso será muito útil na sua carreira como programador.

---

## Sintaxe do array

Existe mais de uma forma de criar uma array no JavaScript, com o método construtor Array() e também de uma forma mais direta e melhor que veremos, você pode utilizar a forma que achar melhor, mas deve saber da diferença, e já adiantando, você irá usar mais a forma direta de criar arrays, e fica como dica: use a forma mais simples mesmo, vejamos como funciona a criação de um novo array no JavaScript:

```
const meuArray = [];
const outroArray = new Array();
```

Bem simples não é mesmo? Vamos dar um console.log() para ver o que temos em mãos:

```
console.log(meuArray); // -> []
console.log(outroArray); // -> []
```

Dá na mesma usar um ou outro nesse caso, e podemos ainda criar esses arrays já colocando algo dentro, vamos fazer um teste:

```
const meuArray = [100];
const outroArray = new Array(100);

console.log(meuArray); // -> [ 100 ]
console.log(outroArray); // -> [ <100 empty items> ]
```

Veja que usando o construtor de arrays e passando 100 como parâmetro o que acontece é que o JavaScript cria um array com 100 posições vazias, e da forma mais simples o que ocorre é a criação de uma array com um valor numérico dentro.

---

Vimos então que podemos iniciar arrays vazios e também com valores, vamos ver mais exemplos abaixo de arrays sendo criados já com valores dentro e também vamos observar que um array pode conter itens de tipos diferentes, o que não é muito comum de vermos por ai, mas é possível,  mas no geral um array contém itens todos do mesmo tipo:

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

const arrayDeAnimais = ['Urso', 'Macaco', 'Tigre', 'Tartaruga', 'Crocodilo'];

const arrayDeObjetos = [
    { nome: 'Lucas', idade: 23 },
    { nome: 'Otávio', idade: 22 },
    { nome: 'Gabriel', idade: 18 },
];

// Os dois casos a seguir não são muito corriqueiros, pois, é bastante dificil de tratar suas iterações

const arrayDeArrays = [
    [1, 2, 3],
    ['lucas', 'otavio', 'gabriel'],
    [
```

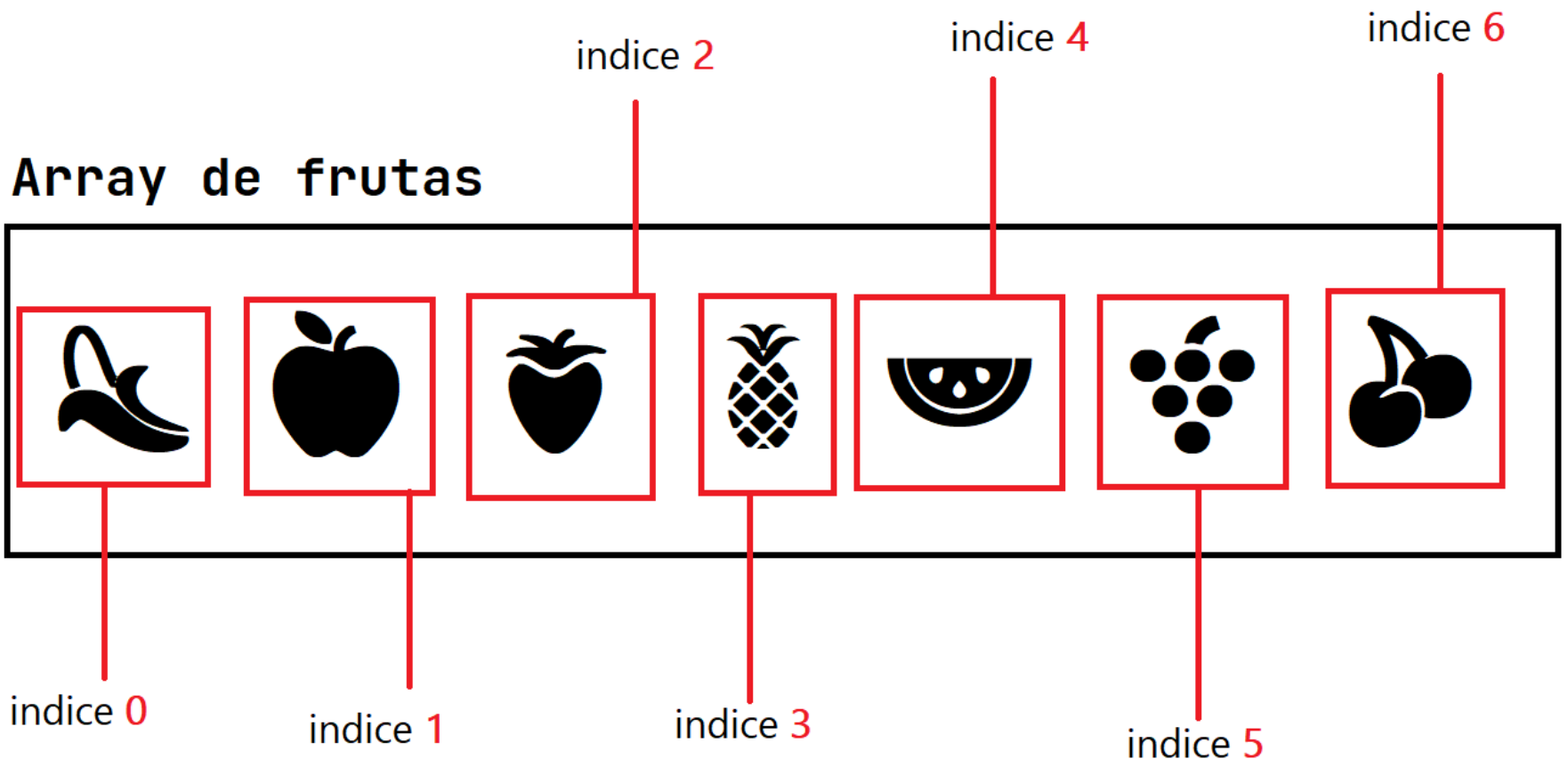
```
        { nome: 'Lucas', idade: 23 },
        { nome: 'Otávio', idade: 22 },
        { nome: 'Gabriel', idade: 18 },
    ]
];

const arrayMisto = [
    10, 20, 'String', 'Outra string', { objeto: 'objeto' }, ['array', 'de', 'string']
];
```

---

## O que é o indice de um item dentro do array?

O índice de um item dentro do array nada mais é do que a sua posição dentro do array em questão, observe:




> Sempre lembre que o primeiro item do array sempre tem o indice ZERO (0), não confunda primeiro item com indice 1.

---

## Como podemos acessar os itens dentro dos arrays?

Como vimos, cada item dentro de um array tem um índice, portanto podemos acessar esses itens à partir do seu índice, veja como é simples:

```
const animais = ['Urso', 'Macaco', 'Tigre', 'Tartaruga', 'Crocodilo'];

const animal1 = animais[0];

console.log(animal1); // -> Urso
```

E se em um array muito grande eu sei o item desejado mas não sei o índice, como faço?

```
const animais = ['Urso', 'Macaco', 'Tigre', 'Tartaruga', 'Crocodilo'];

console.log(animais.indexOf('Tigre')); // -> 2
```

Existe o método .indexOf() dos arrays onde passamos um item que exista no array e ele nos devolve a posição desse item em questão dentro do array, esse é um dos vários métodos de arrays que vamos aprender, outro método bem simples e interessante de aprender desde já é o length:

```
const animais = ['Urso', 'Macaco', 'Tigre', 'Tartaruga', 'Crocodilo'];

console.log(animais.length); // -> 5
```

```

```

o **.length** nos retorna a quantidade de itens que um array possui.

## Métodos de arrays

> Quando temos um array em mãos podemos usar métodos que os arrays nos disponibilizam para fazer manipulações, isso existe em todas as linguagens que trabalham com arrays, vamos aprender e entender os existentes para o JavaScript.

## Inserindo itens dentro de um array com .push e .unshift

Esse é um dos mais utilizados, serve para inserir novos itens ao final de um array.

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.push('item 6');

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5', 'item 6' ]
```

Perceba que o .push() insere um item ao final do array, porém existe um método chamado .unshift() que insere um item no inicio de um array:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.unshift('item 6');

console.log(meuArray);
// -> [ 'item 6', 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]
```

Note que o 'item 6' foi adicionado no array, porém ao invés de ser adicionado ao final do array como acontece com o .push(), ele foi adicionado no início.

Existem outras maneiras de adicionarmos itens dentro de arrays, porém não são muito práticas:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray[5] = 'item 6';

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5', 'item 6' ]
```

O jeito que foi feito é algo como: nessa posição do array quero colocar esse item.

E no exemplo acima foi colocado em uma posição do array que ainda não existia, porém se colocarmos em uma posição já existente o item dessa posição será removido e substituido pelo novo item:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray[2] = 'item 6';

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 6', 'item 4', 'item 5' ]
```

Observe que não é algo muito prático, por isso existem os métodos .push() e .unshift() para facilitar a adição de novos itens nos arrays.

---

Então resumindo a adição de itens em arrays, podemos usar os métodos push e unshift, o .push() adiciona um item ao final do array e o .unshift() no início do array, a sintaxe é bastante simples:

```
array.push(item);

// ou

array.unshift(item);
```

---

## Como remover itens dos arrays?

Vimos como adicionar itens em um array, mas como podemos fazer para remover itens lá de dentro? Existem métodos para isso? Sim, e vamos conhecê-los agora.

Temos 2 métodos equivalentes ao .push() e ao .unshift(), no caso, métodos que fazem o contrário desses 2 citados, o método .pop() remove o último item de um array, e o método .shift() remove o primeiro item de um array, vejamos:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.pop();
console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4' ]
```

Sem muito mistério, o .pop() apenas remove o último item do array, vamos ver agora o .shift():

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.shift();
console.log(meuArray);
// -> [ 'item 2', 'item 3', 'item 4', 'item 5' ]
```

Como esperado, agora o primeiro item é que foi removido, muito interessante, mas e se quisermos remover itens específicos? Para isso existem outros métodos que veremos agora.

---

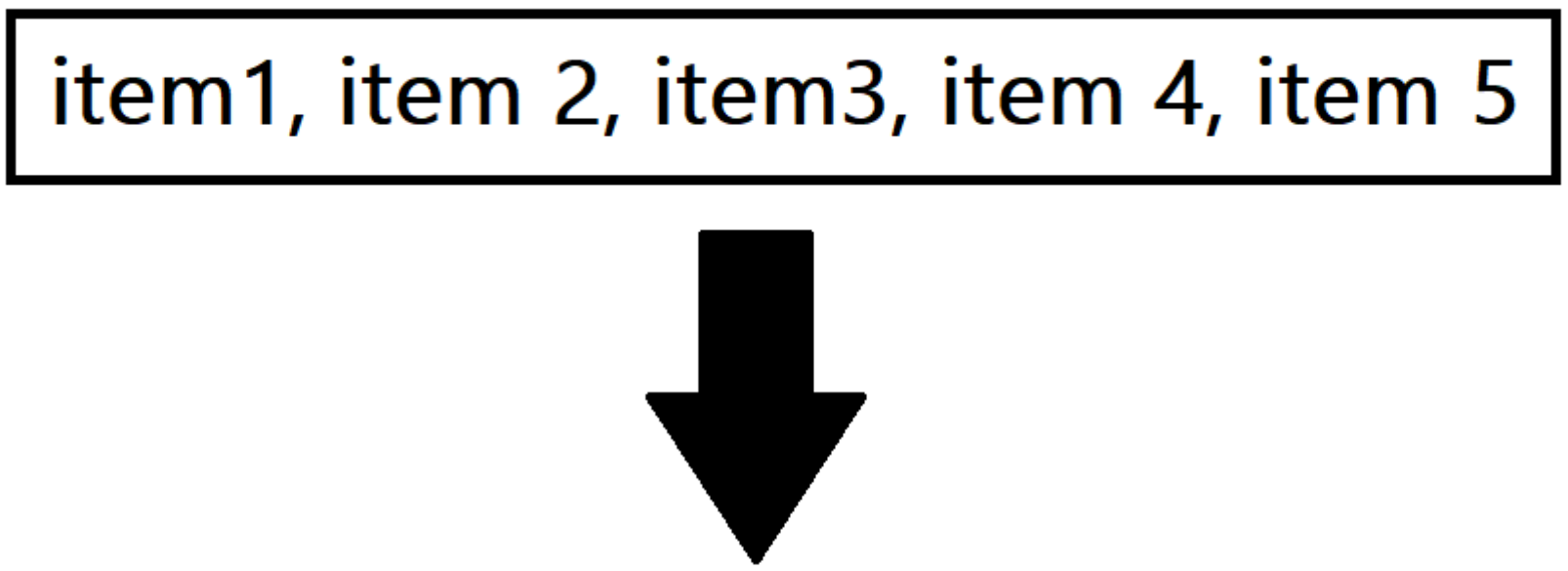
Temos o método chamado .splice() dentro do JavaScript, nele passamos 2 parâmetros, a posição da qual desejamos começar a exclusão de itens e quantos itens queremos excluir, veja como funciona:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.splice(1, 2); // -> a partir da posição 1, remova 2 itens
console.log(meuArray);
// -> [ 'item 1', 'item 4', 'item 5' ]
```

Não tem muito segredo, passamos a posição de início da exclusão, a exclusão inclui o item inicial passado como parâmetro.




Podemos unir o .splice() junto do .indexOf() para excluir itens mais específicos, veja como é interessante e bastante útil essa abordagem

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

let item_remove = meuArray.indexOf('item 3')

meuArray.splice(item_remove, 1);
console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 4', 'item 5' ]
```

Bem simples e extremamente útil.

---

O splice também pode ser utilizado para adicionar itens nos arrays, e para isso precisamos passar mais alguns parâmetros, veja um exemplo:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.splice(0, 0, 3, 3, 3);
console.log(meuArray);
// -> [ 3, 3, 3, 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]
```

**meuArray.splice(0, 0, 3, 3, 3)** → **0, 0** quer dizer "a partir do indice 0 não remova nada" e **3, 3, 3** são os itens que devem ser adicionados ali.

---

O .splice() altera os itens do array, porém existe um método chamado .slice() que não altera o array e ainda cria um novo array com os itens que queremos selecionar, caso tenha ficado confuso não se preocupe, vamos ver direitinho como funciona:

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
```

```
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

meuArray.slice(0, 2);
console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]
```

Ao usar o .slice() diferentemente de como usamos o .splice() nada ocorre com o array, então vamos utilizar a ajuda de uma variável para ver o poder do .slice():

```
const meuArray = ['item 1', 'item 2', 'item 3', 'item 4', 'item 5'];

console.log(meuArray);
// -> [ 'item 1', 'item 2', 'item 3', 'item 4', 'item 5' ]

const recorte = meuArray.slice(0, 2);
console.log(recorte);
// -> [ 'item 1', 'item 2' ]
```

Agora sim faz sentido, o .slice() faz uma espécie de **Ctrl + C** numa parte do array.

---

> Vamos agora conhecer outros métodos para arrays que serão extremamente úteis no seu dia a dia como desenvolvedor.

---

## .forEach()

Esse método é um dos mais usados para percorrer ou iterar itens de um array, e para cada item algo acontece, ou seja, percorremos o array executando algo para cada item de dentro desse array em questão, observe

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 18 }
];

pessoas.forEach(function(pessoa) {
    console.log(`${pessoa.nome} tem ${pessoa.idade} anos`);
});
```

Como resultado do código acima temos o seguinte:

```
Lucas tem 23 anos
Otávio tem 23 anos
Gabriel tem 18 anos
```

**pessoa** é uma variável (parâmetro) criada para ser usada dentro do .forEach(), que no caso representa cada item dentro do array, e no exemplo acima ainda poderíamos fazer criando outras variáveis dentro do .forEach()

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 18 }
];

pessoas.forEach(function(pessoa) {

    let nome = pessoa.nome;
    let idade = pessoa.idade;

    console.log(`${nome} tem ${idade} anos`);
});
```

Outro grande lance sobre o .forEach() é que podemos usar mais 2 parametros: o indice e o próprio array, vamos ver o que temos dentro desses parametros

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
```

```
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 18 }
];

pessoas.forEach(function(pessoa, indice, array) {
    console.log(pessoa); // imprime uma pessoa a cada loop
    console.log(indice); // imprime um indice por vez (0, 1, 2)
    console.log(array); // imprime o array de pessoas inteiro
});
```

É mais comum o uso de apenas um parâmetro, que é o que representa cada item dentro do array a cada loop, mas caso um dia precise usar o indice dos itens de dentro do array dentro do .forEach() ou o próprio array dentro do .forEach() você já sabe como fazer.

---

## .map()

Esse método você pode até achar que funciona como o .forEach(), o que não é mentira, mas o .map() tem o diferencial que pode criar um novo array com base no array percorrido sem fazer alterações nesse array em questão, mas sim com as alterações no novo array retornado, veja como é simples

```
const pessoas = [
    { nome: "Lucas", idade: 23, saldo: 0 },
    { nome: "Otávio", idade: 23, saldo: 0 },
    { nome: "Gabriel", idade: 18, saldo: 0 }
];

const pessoasRicas = pessoas.map(function(pessoa) {
    pessoa.saldo = 10000000;

    return pessoa;
});

console.log(pessoasRicas);
// [
//     { nome: 'Lucas', idade: 23, saldo: 10000000 },
//     { nome: 'Otávio', idade: 23, saldo: 10000000 },
//     { nome: 'Gabriel', idade: 18, saldo: 10000000 }
// ]


// ou

const pessoas = [
    { nome: "Lucas", idade: 23, saldo: 10000000 },
    { nome: "Otávio", idade: 23, saldo: 0 },
    { nome: "Gabriel", idade: 18, saldo: 0 }
];

const pessoasRicas = [];

pessoas.map(function(pessoa){
    if(pessoa.saldo >= 10000000){
        pessoasRicas.push(pessoa)
    }
})

console.log(pessoasRicas)
```

Então quando usar o .forEach() ou .map() depende do que você quer fazer, se quer apenas percorrer cada item fazendo algo com cada item use o .forEach(), caso queira um novo array com base no array original mas em alterar o array original use o .map().

---

## .find()

Se formos traduzir do inglês a palavra "find" teremos "achar" como resposta, e é bem isso que esse método faz, ele "acha" algo para nós com base em uma pequena função passada por parâmetro, porém ele nos retorna o primeiro item encontrado, para ficar mais claro veja o exemplo

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 }
];

const menorDeIdade = pessoas.find(function(pessoa) {
    return pessoa.idade < 18;
```

```
});

console.log(menorDeIdade);
// -> { nome: 'Gabriel', idade: 17 }
```

Mas e se tivéssemos mais ocorrências de pessoas menores de idade, o que será que acontece?

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 },
    { nome: "Amanda", idade: 17 }
];

const menorDeIdade = pessoas.find(function(pessoa) {
    return pessoa.idade < 18;
});

console.log(menorDeIdade);
// -> { nome: 'Gabriel', idade: 17 }
```

Ainda sim, apenas o "Gabriel" seria retornado, e se quisermos que todas as pessoas menores de idade fossem mostradas? Para isso precisamos usar outro método, o .filter().

---

## .filter()

O .filter() faz praticamente o mesmo que o .find(), com a diferença de que trás todas as ocorrências do que queremos dentro de um array, observe

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 },
    { nome: "Amanda", idade: 17 }
];

const menorDeIdade = pessoas.filter(function(pessoa) {
    return pessoa.idade < 18;
});

console.log(menorDeIdade);
// -> [ { nome: 'Gabriel', idade: 17 }, { nome: 'Amanda', idade: 17 } ]
```

Mesmo se tivermos apenas uma ocorrência essa única ocorrência será retornada dentro de um array.

---

## .findIndex()

Esse faz o mesmo que o .find(), porém ao invés do item ele nos retorna o indice do item

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 }
];

const menorDeIdade = pessoas.findIndex(function(pessoa) {
    return pessoa.idade < 18;
});

console.log(menorDeIdade);
// -> 2
```

---

## .every()

Esse método verifica se todos os itens de um array passam em uma determinada condição, veja

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 18 }
];
```

```
const menorDeIdade = pessoas.every(function(pessoa) {
    return pessoa.idade >= 18;
});

console.log(menorDeIdade);
// -> true
```

Nesse caso acima todas as pessoas tem 18 anos ou mais, agora veja outro exemplo

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 }
];

const menorDeIdade = pessoas.every(function(pessoa) {
    return pessoa.idade >= 18;
});

console.log(menorDeIdade);
// -> false
```

Temos uma pessoa menor de idade, então por isso a condição não é verdadeira para todos, por isso temos false como retorno.

## .some()

Esse faz algo bem parecido com o .every(), mas a diferença é que se pelo menos um dos itens satisfazerem a condição o retorno já será true

```
const pessoas = [
    { nome: "Lucas", idade: 23 },
    { nome: "Otávio", idade: 23 },
    { nome: "Gabriel", idade: 17 }
];

const menorDeIdade = pessoas.some(function(pessoa) {
    return pessoa.idade >= 18;
});

console.log(menorDeIdade);
// -> true
```

## .sort()

O .sort() serve para fazer a ordenação os itens dentro do array, porém quase sempre as pessoas usam o .sort() da forma errada, sem passar nenhum parâmetro, a ordenação até irá acontecer mas de forma não tão organizada, para usar o sort corretamente devemos fazer comparações, vamos usar como exemplo um array de pessoas e ordenar de diferentes maneiras, assim quando você precisar já saberá o que fazer, para usar o .sort() da forma correta utilizamos 2 parametros para comparar os itens de dentro do array

```
const pessoas = [
    { nome: "Lucas", idade: 23, saldo: 200 },
    { nome: "Gabriel", idade: 17, saldo: 4200 },
    { nome: "Otávio", idade: 23, saldo: 2500 },
    { nome: "Marcela", idade: 59, saldo: 100 },
];

const pessoasMaisRicas = pessoas.sort(function (a, b) {
    if (a.saldo > b.saldo) {
        return -1;
    } else if (a.saldo < b.saldo) {
        return 1;
    }
    return 1;
});

console.log(pessoasMaisRicas);
```

Nesse exemplo apresentado temos o que mais se aproxima do que você irá precisar um dia, que é ordenar arrays de objetos de acordo com propriedades específicas, por exemplo ordenar uma lista de pessoas por ordem alfabética, por idade, ou então uma

lista de qualquer coisa por quantidade em ordem crescente ou decrescente, o que muda é apenas a ordem dos sinais de maior e menor dentro do if.

---

## .reverse()

Esse método inverte o array, deixando ele em ordem reversa, bem simples, veja um exemplo:

```
const numeros = [1, 2, 3, 4, 5];
console.log(numeros);
// -> [ 1, 2, 3, 4, 5 ]

numeros.reverse();
console.log(numeros);
// -> [ 5, 4, 3, 2, 1 ]
```

---

## .concat()

*".concat() cria um novo array unindo todos os elementos que foram passados como parâmetro, na ordem dada, para cada argumento e seus elementos (se o elemento passado for um array). .concat() não altera a si mesmo ou a qualquer um dos argumentos passados, apenas providencia um novo array contendo uma cópia de si mesmo e dos argumentos passados" - MDN Web Docs*

```
const animais1 = ['Leão', 'Tubarão', 'Galinha', 'Jacaré', 'Tucano', 'Coala'];
const animais2 = ['Tigre', 'Onça', 'Cachorro', 'Papagaio', 'Raposa', 'Rato'];

const animais = animais1.concat(animais2)

console.log(animais);
// [
//     'Leão', 'Tubarão',
//     'Galinha', 'Jacaré',
//     'Tucano', 'Coala',
//     'Tigre', 'Onça',
//     'Cachorro', 'Papagaio',
//     'Raposa', 'Rato'
// ]
```

---

## .reduce()

Executa uma função de retorno de chamada em cada item da matriz que é invocado. Pode aceitar até 4 parâmetros:

- "acumulador" → acumula valores de retorno da chamada;
- "valor atual" → elemento atual que está sendo processado pela matriz;
- índice → é o índice de cada item (é opcional);
- matriz → também é um parâmetro opcional;

```
const numeros = [3, 6, 9, 12];

const redutor = (valorAnterior, valorAtual) => valorAnterior + valorAtual;

console.log(numeros.reduce(redutor));
// Retorna: 30

console.log(numeros.reduce(redutor, 10));
// Retorna: 40
```

---

## .entries()

Retorna um Array Iterado que possui os pares de `[chave, valor]` para cada índice do array. Veja o seguinte exemplo:

```
for (const [indice, valor] of ['Valor1', 'Valor2'].entries()) {
  console.log(indice + ' -> ' + valor);
}

/* Retorna:
  0 -> Valor1
```

```
  1 -> Valor2
*/
```

Usamos um laço `for-of` que nos mostra via console o índice do elemento do array e seu respectivo valor.

---

## .fill

Responsável por preencher todos os elementos do array com um valor estático. Pode ter valor inicial e valor final, sendo opcionais. Caso o valor inicial seja negativo, é considerado um valor igual a `tamanho do array + valor inicial negativo` e, caso o valor final seja negativo, teremos a mesma ideia `tamanho do array + valor final negativo`. Exemplos:

```
let numeros = [3, 6, 9, 12];

numeros.fill(5);         // Retorna: [ 5, 5, 5, 5 ]
numeros.fill(5, 2);      // Retorna: [ 3, 6, 5, 5 ]
numeros.fill(5, 2, 3);   // Retorna: [ 3, 6, 5, 12 ]
numeros.fill(5, -1);     // Retorna: [ 3, 6, 9, 5 ]
numeros.fill(5, -3, 3);  // Retorna: [ 3, 5, 5, 12 ]
numeros.fill(5, 1, -2);  // Retorna: [ 3, 5, 9, 12 ]
```

Para melhor entendimento, pense que a sintaxe desse método é:

```
arrayQualquer.fill(valorEstatico, posicaoInicial, posicaoFinal)
```

---

## .flatMap()

Retorna um novo array formado pela aplicação de uma determinada função de retorno de chamada a cada elemento do array e por fim, deixa o resultado no mesmo nível. Um pouco confuso não é mesmo? Mas vamos ver um exemplo para entendermos:

```
let numeros = [3, 6, 9, 12];

const numbers1 = numeros.flatMap(x => [x, x * 2]);
console.log(numbers1);
/* Retorna:
[
  3,  6,  6, 12,
  9, 18, 12, 24
]
*/

const numbers2 = numeros.reduce((valorEstatico, x) => valorEstatico.concat([x, x * 2]), []);
console.log(numbers2);
/* Retorna:
[
  3,  6,  6, 12,
  9, 18, 12, 24
]
*/
```

Um ótimo motivo para usá-lo é esse descrito no exemplo: em vez de usar .reduce() e .concat() juntos, podemos utilizar somente o .flatMap().

Perceba que ambos os modos retornam exatamente os mesmos resultados.

---

Como recomendação, caso queira se aprofundar ainda mais nesse assunto, segue o link da documentação:

JavaScript Array Reference

The Array object is used to store multiple values in a single variable: Array indexes are zero-based: The first element in the array is 0, the second is 1, and so on. For a tutorial about Arrays, read our JavaScript Array Tutorial.

https://www-w3schools-com.translate.goog/jsref/jsref_obj_array.asp?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=nui,sc,elem


---

Esse métodos citados acima são os principais, e os que você mais vai lidar no dia a dia provavelmente serão esses:

- .forEach()
- .map()

- .filter()
- .splice()
- .slice()
- .sort()
- .indexOf()

> Certifique de que tenha entendido bem como esses carinhas funcionam, eles serão de grande ajuda no seu dia a dia, saber manipular estruturas como arrays de objetos é extremamente importante e extremamente comum.

---

***Ps.: Além dos métodos convencionais de Array temos uma biblioteca de javascript que tem algumas funções bem bacanas sobre consumo de cadeias de dados, que é o Lodash. Ela facilita o trabalho com o Javascript em geral, como com Arrays, Objetos, Strings e etc... Recomendamos você a conhecê-la:***

Lodash
A JavaScript utility library delivering consistency, modularity, performance, & extras.
https://lodash.com/