Funções pre-definadas com arrays.

* concat()
  + Junta dois arrays, colocando o array passado como argumento, logo depois do primeiro. Em português essa operação é conhecida como concatenação.
  + Não altera o array no qual foi chamado, então precisamos salvar esse resultado em um novo array.
* filter()
  + Retorna uma lista contando todos os elementos que passaram em um teste, ou seja, uma função escrita por nós.
  + Não altera o array onde foi chamado, então precisamos salvar esse resultado em um novo array.
* find()
  + Funciona de forma parecida com o filter, porém retorna apenas o primeiro valor que satisfizer o teste, podendo ser uma string ou um número.
* findIndex()
  + Funciona igual o find(), mas retorna o índice em vez do elemento, possibilitando usá-lo em outras partes do código.
* lastIndexOf()
  + É igual o findIndex() porém começa do último elemento, não no primeiro.
* forEach()
  + Executa uma função em cada elemento do array de forma individual.
  + Não altera o array original e nem retorna um valor, deixando esse trabalho a cargo da função escolhida.
* pop()
  + Retira o último elemento do array.
  + Altera o array original removendo o elemento.
* shift()
  + Retira o primeiro elemento do array.
  + Altera o array original removendo o elemento e trocando o índice de todos os elementos para um a menos do que eram, o índice 1 passa a ser o 0, o 2 passa a ser o 1, e assim por diante.
* push()
  + Adiciona o elemento passado como parâmetro do array, porém na última posição.
  + Altera o array original com o novo valor.
* unshift()
  + Funciona igual ao push(), porém adiciona na primeira posição e acaba trocando o índice de todos os elementos.
  + Altera o array original com o novo valor.
* reduce()
  + Utiliza uma função definida pelo usuário em cada um dos elementos, guardando o resultado em uma variável que pode ser acessada dentro da função que foi definida, retornando um único valor no final, reduzindo o array para um único valor.
* reduceRight()
  + Funciona igual o reduce() porém começa do final do array e segue até o início.
* reverse()
  + Inverte a ordem dos elementos do array, então o primeiro vira o último, o segundo o penúltimo e assim por diante.
* slice()
  + Copia uma parte do array para outro array.
* sort()
  + Organiza o array de acordo com a classificação Unicode, onde os números vêm antes das letras, porém não funciona corretamente para números, onde temos que definir uma função que irá auxiliar o comando.
* splice()
  + Consegue remover, um ou mais elementos consecutivos caso o segundo parâmetro tenha um valor maior que 0, e incluir um ou mais elementos a partir de um índice escolhido.

**Nesta aula vimos:**

* **As características principais de arrays:**
  + Um array é uma lista ordenada de dados. Você aprendeu a sintaxe de como criar um array e acessar seus valores.
* **Os conceitos de *elemento* e *índice*:**
  + Elementos são os valores que um array guarda, e índice é o identificador único e numérico de cada elemento do array.
* **Como utilizar a propriedade .length:**
  + Todos os arrays do JavaScript possuem essa propriedade, que indica a quantidade de elementos do array.
* **Como utilizar os métodos .push() e .pop():**
  + Vimos como realizar as primeiras alterações em um array, como adicionar um novo item ao final ou remover o último item.

**Nesta aula aprendemos como:**

* Separar um array em partes com slice() (utilizamos para obter dois novos arrays a partir de um).
* Remover e incluir novos elementos em um array com splice(); sendo que esse método permite remover elementos de qualquer posição do array. Além disso, podemos incluir um novo elemento no local dos removidos.
* Concatenar elementos em um único array com concat(); sendo que essa é uma forma facilitada de juntar dois arrays.
* Trabalhar com arrays de duas dimensões. Vimos que uma lista também pode guardar outras listas! Para acessar elementos das listas mais internas, devemos informar primeiro os índices das mais externas e depois os das mais internas, como por exemplo com funcionarios[0][2].

**Nesta aula vimos como:**

* **Utilizar os métodos includes() e indexOf():**
  + O includes() retorna true ou false, dependendo se o valor passado como parâmetro está presente ou não no array. O indexOf() retorna a posição no array do valor passado como parâmetro. Utilizamos esses dois métodos em conjunto para primeiro verificar se um aluno estava presente em uma lista, e depois obter sua posição para buscar sua nota em outra lista.
* **Desestruturar um array:**
  + Você aprendeu a sintaxe de desestruturação, disponível desde o ES6 (EcmaScript 6), que nos permite criar várias novas variáveis ao mesmo tempo a partir dos elementos de um array.
* **Utilizar a estrutura for e entender seu funcionamento:**
  + O for é constituído de três expressões: criação da variável de controle; condição de execução; modificação da variável de controle. Essas três expressões juntas definem quantas vezes o bloco do for será executado.
* **Utilizar a estrutura for of e entender a diferença em relação ao for:**
  + O for of sempre é utilizado para percorrer todos os elementos de um array, do início ao fim. Para esses casos, o for of é mais conciso que o for.

# Para saber mais: for...of vs callbacks

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/javascript-arrays/task/114499/next)

Até agora vimos várias formas de fazer o que parece ser a mesma coisa: for, for…of, forEach()... Com tantas opções, como posso escolher qual usar no meu código?

Antes, vamos dar uma olhada em cada um deles:

## for

A forma mais “clássica” de se efetuar um loop em JavaScript e em várias outras linguagens, o for é muito conveniente pois pode ser utilizado com qualquer tipo de **iterável** e é construído de uma forma que deixa muito claro quais são todas as “fases” de cada loop - **também chamamos um loop de laço de repetição ou de iteração**.

**O que é um “iterável”?** Além de arrays, strings, sets (conjuntos) e maps (mapas ou dicionários) são considerados **iteráveis**. Não vamos falar dos dois últimos tipos neste curso, mas se você tiver interesse em saber mais sobre conjuntos, dicionários e outras estruturas de dados, pode dar uma olhada neste [artigo](https://www.alura.com.br/artigos/estruturas-de-dados-introducao); o que precisamos saber agora é que um **iterável**, aqui, representa uma sequência de elementos que pode ser percorrida (ou seja, **iterada**) utilizando ferramentas próprias para isso. É importante fazer uma distinção entre array e iterável, pois nem todo método que funciona em um array vai funcionar em outros iteráveis - veremos isso em seguida.

const lista = [1, 2, 3, 4, 5];

for (let indice = 0; indice < lista.length; indice++) {

console.log(lista[indice]);

}COPIAR CÓDIGO

O que queremos dizer com “deixar claro as fases da iteração” pode ser visto no exemplo acima: a expressão entre parênteses () após a palavra-chave for. Dentro da expressão temos:

* uma variável contadora, que é criada antes do início do laço (let indice);
* uma expressão de teste (indice < lista.length), que é executada antes de cada iteração e que verifica se o código dentro do bloco {} deve ou não ser executado;
* por último, uma expressão que é executada ao final de cada laço, normalmente um incremento (++) ou decremento (--) da variável contadora.

Ou seja: como as fases de cada laço são declaradas de forma explícita, elas também podem ser alteradas conforme a necessidade do código, o que faz com que o for seja muito versátil e possa ser utilizado em casos específicos, quando os outras formas de sintaxe mais reduzida (como os que vamos ver em seguida) não atendem. O laço pode ser decremental ao invés de incremental (percorrer um array de trás para frente), a variável contadora pode receber outro valor como let indice = 2 ou let indice = outraVariavel + 1 (desde que esse valor seja avaliado como número), a condição de teste pode ser o tamanho do array como no exemplo, ou outro (indice < lista.length - 1) e o incremento ou decremento ao final do laço pode receber qualquer operador aritmético ou ser o resultado de uma operação, como indice = indice + 2 ou indice \*= 2.

É possível, inclusive, usar o for para executar algoritmos mais complexos, que também não seriam possíveis com outros métodos, por exemplo:

const lista = [1, 2, 3, 4, 5];

for (let i = 0, j = 0; i < lista.length; i++, j++) {

console.log(lista[i] + j);

}

//1

//3

//5

//7

//9COPIAR CÓDIGO

Ainda há outras formas de se trabalhar com as condições do for, que você pode conferir na [documentação sobre for no MDN](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/for).

## for…of

Adicionado às funcionalidades do JavaScript na versão ES6, é um tipo de laço de repetição diferente do for tradicional, embora utilize a mesma palavra-chave:

const lista = [1, 2, 3, 4, 5];

let soma = 0;

for (let elemento of lista) {

soma += elemento;

}

console.log(soma) //15COPIAR CÓDIGO

O for…of pode ser executado utilizando qualquer tipo de **iterável**: além de arrays, strings, sets (conjuntos) e maps (mapas ou dicionários) são considerados **objetos iteráveis**. Não vamos falar dos dois últimos tipos neste curso, mas se você tiver interesse em saber mais sobre conjuntos, dicionários e outras estruturas de dados, pode dar uma olhada neste [artigo](https://www.alura.com.br/artigos/estruturas-de-dados-introducao); o que precisamos saber agora é que um **iterável**, aqui, representa uma sequência de elementos que pode ser percorrida (ou seja, **iterada**) utilizando o for…of. É importante fazer esta distinção entre array e iterável, pois nem todo método ou declaração que funciona em um array vai funcionar em outros iteráveis.

Olhando assim, o for…of se parece bastante com o for: temos a declaração de uma variável (let elemento) seguida da palavra-chave of e um identificador do iterável que será percorrido (no caso acima é um array mesmo, lista).

A diferença principal e mais visível entre o for e o for…of é que o anterior dá muito mais controle sobre de que forma o laço de repetição ocorre. Por exemplo, vimos que no for é possível manipular de forma mais fina todas as condicionais; já no for…of as condições são mais “fixas”: elemento se refere a cada elemento (ou item) do array e o loop sempre ocorre de forma sequencial, do primeiro elemento até o último.

Em contrapartida, a sintaxe simplificada e mais “legível” para pessoas faz com que o uso do for…of seja mais prático do que o for, pois laços de repetição mais simplificados, que percorrem um iterável do primeiro ao último elemento e executam o código do bloco a cada iteração, são muito mais corriqueiros.

## forEach()

Ao contrário de for e for…of, o forEach() é um **método do objeto Array**.

O que é um **método**? Em programação, chamamos de método uma função que é propriedade de um objeto. A [documentação do MDN](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Functions/Method_definitions) traz alguns exemplos de métodos, mas você vai poder conferir com mais detalhes o que são objetos e suas propriedades quando fizer o curso de [fundamentos do JavaScript: Objetos](https://cursos.alura.com.br/course/fundamentos-javascript-objetos). Por agora, basta saber que quando dizemos que determinada função - por exemplo, forEach() é um **método de array**, significa que esta função foi desenvolvida para trabalhar apenas com esse tipo de dado - no caso, com arrays.

O forEach(), assim como outros métodos de array que vimos ou ainda vamos ver durante esta aula, também cumprem o papel de iterar arrays, porém com funcionalidades e retornos bem definidos. No caso do forEach(), apesar da sintaxe bem diferente, podemos utilizar este método como o for ou o for…of, pois ele vai executar as instruções que forem passadas para cada elemento iterado, sem retornar nenhum dado.

O forEach(), assim como alguns outros [métodos de array do JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) que estamos vendo no curso, utiliza a abordagem funcional de **funções callback** para executar o código necessário a cada elemento iterado no laço:

const lista = [1, 2, 3, 4, 5];

let soma = 0;

lista.forEach(numero => soma += numero);

console.log(soma) //15COPIAR CÓDIGO

Como visto acima, o método forEach() recebe uma função como parâmetro, e esta função por sua vez utiliza como parâmetro cada um dos elementos do array. A lógica interna do forEach() vai manejar a iteração do primeiro ao último item do array e fazer com que o código definido dentro da função callback (no caso, apenas um incremento soma += numero) seja executado a cada iteração.

O mesmo código também poderia ser escrito da seguinte forma, um pouco mais extensa porém mais legível:

lista.forEach(numero => {

soma += numero

});COPIAR CÓDIGO

Ou ainda, utilizando a palavra-chave function ao invés de arrow functions; essa forma de escrita não é usual, pois quando usamos funções callback o padrão adotado é o de arrow functions, mas serve para o propósito de legibilidade:

lista.forEach(function (numero) {

soma += numero

});COPIAR CÓDIGO

Assim, vemos que o forEach() não **retorna** nenhum tipo de valor, apenas executa o que está dentro do bloco da função callback, assim como for e for…of. Porém também não é possível modificar a forma que a iteração será feita (é sempre do primeiro elemento do array ao último) e não há forma de interromper o laço dada alguma condição, como o break funcionaria para um for normal.

O forEach() também não pode ser utilizado com qualquer iterável, apenas com arrays e também não pode ser utilizado de forma assíncrona - um tema um pouco mais avançado e que não trabalharemos nesse curso, mas que você pode conferir [neste artigo](https://www.alura.com.br/artigos/async-await-no-javascript-o-que-e-e-quando-usar) se tiver curiosidade pra começar a aprender.

## E quando devo utilizar um ou outro?

A resposta, como em muitas coisas na programação, é **depende**!

Atualmente, em termos de performance - ou seja, qual código é executado de forma mais rápida pelo computador e consumindo menos recursos de memória - tanto os laços com for como o forEach() não tem diferenças significativas, embora ainda possam ocorrer exceções, como no caso de versões muito antigas de navegadores e/ou manipulação de arrays muito longos. Mas no momento em que desenvolvemos este curso a questão de performance não é tão definidora na maior parte dos casos, pois os interpretadores do JavaScript já evoluíram para trabalhar tão bem com funções callbacks quanto com iteradores.

O forEach() utiliza callbacks assim como map(), filter() e alguns outros métodos de array, o que facilita a utilização desses métodos em conjunto ou a troca de um para outro dependendo da necessidade do código ou de uma refatoração. Alguns guias de estilo de código desenvolvidos pela comunidade, como o famoso [guia do AirBNB](https://github.com/airbnb/javascript#iterators-and-generators), indica preferencialmente o uso do forEach() no lugar do for…of para manter a consistência no estilo do código.

O for…of, mais recente que o forEach(), não é um método de array e sim um **iterador**, assim como o for. Ou seja, vai trabalhar com outros tipos de dados iteráveis além de arrays, com uma sintaxe mais simplificada que a do for.

Como em ambos os casos não há retorno, somente a execução do código que está dentro do bloco - os colchetes {} do for…of ou a função callback do forEach(), você pode utilizar as ferramentas da seguinte forma:

* forEach() para trabalhar com arrays da forma mais corriqueira - percorrendo do primeiro ao último elemento, sem alterar a condição de parada, e também para manter a coesão do estilo quando em conjunto com outros métodos de array como map, filter e etc;
* for…of em caso de iteráveis (dicionários, conjuntos e outras estruturas de dados) ou de arrays quando for necessário o uso de programação assíncrona e/ou dar condições de saída do laço (por exemplo, com o uso de break);
* for para casos em que seja necessário manipular de forma mais fina as fases do laço (condição inicial, condição de parada e incremento).