



# Arrays e Estruturas de Repetição

☰ Conhecimentos	
📅 Data da aula	@5 de agosto de 2024
☰ Tipo	Atividade em Classe Aula Expositiva




Serviço Nacional de Aprendizagem Comercial

Departamento Regional de Pernambuco

Habilitação Profissional Técnica em Informática

UC9 - Desenvolver Algoritmos

Thiago Nogueira - Instrutor de Educação Tecnológica

 [thiago.nogueira@senac.pe.br](mailto:thiago.nogueira@senac.pe.br)



[linkedin.com/tdn](https://www.linkedin.com/tdn)



[\(81\) 9 9627-0419](tel:(81)99627-0419)



[@thiagoo.\\_nogueiraa](https://www.instagram.com/thiagoo._nogueiraa)

## Array

Organizar e armazenar dados é um conceito fundamental de programação. Uma forma de organizar os dados na vida real é fazendo listas.

Metas de ano novo:

1. Manter um diário
2. Fazer uma aula de dança
3. Aprender a fazer malabarismos

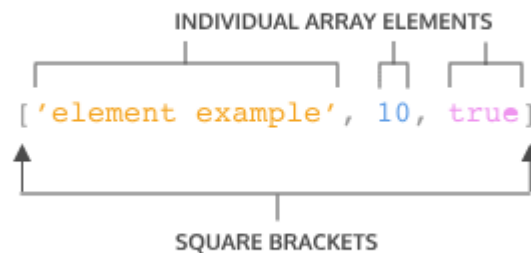
Vamos agora escrever esta lista em JavaScript, como um *array* :

```
let newYearsResolutions = ['Manter um diário', 'Fazer uma aula de dança', 'Aprender a fazer malabarismos']
```

Arrays são a maneira do JavaScript criar listas. Arrays podem armazenar qualquer tipo de dados (incluindo strings , números e booleanos). Assim como as listas, os arrays são ordenados, o que significa que cada item possui uma posição numerada.

## Criando Arrays

Uma maneira de criar um array é usar um *array literal*. Um array literal cria um array colocando os itens entre colchetes `[]`. Os arrays podem armazenar qualquer tipo de dados - podemos ter um array que contém todos os mesmos tipos de dados ou um array que contém diferentes tipos de dados.



Array Exemplo

- O array é representado pelos colchetes `[]` e o conteúdo dentro deles.
- Cada item de dentro de um array é chamado de *elemento*.
- No exemplo, cada elemento dentro da matriz é um tipo de dados diferente.

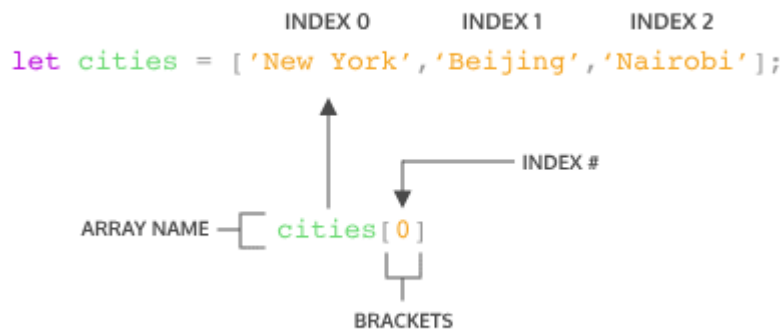
Também podemos salvar um array em uma variável.

```
let newYearsResolutions = ['Keep a journal', 'Take a falconry
```

## Acessando Elementos

Cada elemento em um array possui uma posição numerada conhecida como *índice*. Podemos acessar itens individuais usando seu índice, o que é semelhante a fazer referência a um item em uma lista com base na posição do item.

Arrays em JavaScript são *indexadas em zero*, o que significa que as posições começam a contar em `0` vez de `1`. Portanto, o primeiro item de um array estará na posição `0`.



Acessando elementos de um array

- `cities` é uma matriz que possui três elementos.
- Estamos usando a notação de colchetes, `[]` com o índice após o nome do array para acessar o elemento.
- `cities[0]` acessará o elemento no índice `0` do array `cities`. Você pode pensar nisso `cities[0]` como acessar o espaço na memória que contém a string `'New York'`.

Também pode-se acessar caracteres individuais em uma string usando a notação de colchetes e o índice. Por exemplo:

```
const hello = 'Hello World';
console.log(hello[6]);
// Output: W
```

## Atualizar elementos

Depois de ter acesso a um elemento em um array, você pode atualizar seu valor.

```
let estacoes = ['Inverno', 'Primavera', 'Verão', 'Outono'];

estacoes[3] = 'Mormaço';
console.log(estacoes);
//Output: ['Inverno', 'Primavera', 'Verão', 'Mormaço']
```

No exemplo acima, o array `estacoes` contém os nomes das quatro estações. No entanto, decidimos que preferiríamos salvar 'Mormaço' em vez de 'Outono'.

A linha `estacoes[3] = 'Mormaço';` diz ao nosso programa para alterar o item no índice 3 do array `estacoes`.

## Arrays com `let` e `const`

Variáveis declaradas com `let` podem ser reatribuídas.

Variáveis declaradas com a `const` não podem ser reatribuídas. No entanto, os elementos de um array declarado com `const` permanecem mutáveis. O que significa que podemos alterar o conteúdo de um array `const`, mas não podemos reatribuir um novo array ou um valor diferente.

```
// Código Exemplo
let condimentos = ['Ketchup', 'Mostarda', 'Molho de Soja', 'B

const utensilios = ['Garfo', 'Faca', 'Hachi', 'Garfo'];

condimentos[0] = 'Mayo'
console.log(condimentos)
condimentos = ['Mayo']
console.log(condimentos)

utensilios[3] = 'Colher'
console.log(utensilios)
```

## A propriedade `.length`

Uma das propriedades integradas de um array é o `length` que retorna o número de itens no array. Acessamos a propriedade `.length` da mesma forma que fazemos com strings.

```
const metasDeAnoNovo = ['Manter um diário', 'Fazer uma aula d

console.log(metasDeAnoNovo.length);
//Output: 2
```

Quando queremos saber quantos elementos existem em um array, podemos acessar a propriedade `.length`.

## O método .push()

Vamos aprender sobre alguns métodos integrados que facilitam o trabalho com arrays. Esses métodos são chamados especificamente em arrays para tornar tarefas comuns, como adicionar e remover elementos, mais simples.

Um método, `.push()`, nos permite adicionar itens ao final de um array.

```
const itemRastreador = ['item 0', 'item 1', 'item 2'];

itemRastreador.push('item 3', 'item 4');

console.log(itemRastreador);
// Output: ['item 0', 'item 1', 'item 2', 'item 3', 'item 4']
```

Então, como funciona `.push()`?

- Acessamos o `itemRastreador` usando notação de ponto, conectando com um ponto final.
- Então chamamos isso de função. Isso porque `.push()` é uma função que o JavaScript nos permite usar diretamente em um array.
- `.push()` pode receber um único argumento ou vários argumentos separados por vírgulas. Neste caso, estamos adicionando dois elementos: `'item 3'` e `'item 4'`.
- Observe que `.push()` muda o `itemRastreador`. Ele também é chamado de método de array *destrutivo*, pois altera o array inicial.

## O método .pop()

Outro método de array, `.pop()` remove o último item de um array.

```
const novoItemRastreador= ['item 0', 'item 1', 'item 2'];

const removido = novoItemTracker.pop();

console.log(novoItemRastreador);
//Saída: ['item 0', 'item 1']
console.log(removido);
//Saída: item 2
```

- No exemplo acima, chamando `.pop()` ao final do array, remove o último elemento, o `item 2`
- `.pop()` não aceita nenhum argumento, simplesmente remove o último elemento de `novoItemRastreador`.
- `.pop()` retorna o valor do último elemento. No exemplo, armazenamos o valor retornado em uma variável `removido` para ser usada posteriormente.
- `.pop()` é um método que altera o array inicial.

## Mais métodos de arrays

Existem muito mais métodos de array do que apenas `.push()` and `.pop()`. Você pode ler sobre esses métodos de array no documento [JavaScript Array Methods](#).

`.pop()` e `.push()` altere o array no qual eles são chamados. No entanto, há momentos em que não queremos alterar o array original e podemos usar métodos de array sem fazer mutações. Certifique-se de verificar a documentação para entender o comportamento do método que você está usando.

Alguns métodos de arrays disponíveis para desenvolvedores JavaScript incluem: `.join()`, `.slice()`, `.splice()`, `.shift()`, `.unshift()` e `.concat()` entre muitos outros. O uso desses métodos integrados facilita a execução de algumas tarefas comuns ao trabalhar com arrays.

## Arrays Aninhados

Os arrays podem armazenar outros arrays. Quando um array contém outro array, ele é conhecido como *array aninhado*. Por exemplo:

```
const nestedArr = [[1], [2, 3]];
```

Para acessar os arrays aninhados podemos usar a notação de colchetes com o valor do índice, assim como fizemos para acessar qualquer outro elemento:

```
const nestedArr = [[1], [2, 3]];  
  
console.log(nestedArr[1]); // Output: [2, 3]
```

Observe que `nestedArr[1]` irá capturar o elemento no índice 1 que é o array `[2, 3]`. Então, se quisermos acessar os elementos dentro do array aninhado, podemos *encadear* ou adicionar mais notação de colchetes com valores de índice.

```
const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]
console.log(nestedArr[1][0]); // Output: 2
```

## Atividades de Aprendizagem

### Exercício 1

1. Crie um array chamado `fruits` que contenha as strings `"apple"`, `"banana"`, `"cherry"`.
2. Acesse e imprima o primeiro e o terceiro elemento do array no console.

### Exercício 2

1. Crie um array chamado `colors` que contenha as strings `"red"`, `"green"`, `"blue"`.
2. Atualize o segundo elemento para `"yellow"`.
3. Imprima o array atualizado no console.

### Exercício 3

1. Crie um array chamado `animals` usando `const` que contenha as strings `"cat"`, `"dog"`, `"bird"`.
2. Use o método `.push()` para adicionar `"fish"` ao array `animals` e imprima o array.
3. Tente reatribuir o array `animals` a um novo array `["lion", "tiger"]` e observe o que acontece.

### Exercício 4

1. Crie um array chamado `numbers` que contenha os números `1`, `2`, `3`.
2. Imprima o comprimento do array usando a propriedade `.length`.



3. Use o método `.push()` para adicionar o número `4` ao array e imprima o array.
4. Use o método `.pop()` para remover o último elemento do array e imprima o array.

## Exercício 5

1. Crie um array chamado `matrix` que contenha dois arrays: `[1, 2, 3]` e `[4, 5, 6]`.
2. Acesse e imprima o segundo elemento do primeiro array aninhado.
3. Atualize o terceiro elemento do segundo array aninhado para `7` e imprima o array `matrix`.

# Estruturas de Repetição

## Loop for

Em vez de escrever o mesmo código repetidamente, os loops nos permitem dizer aos computadores para repetirem um determinado bloco de código por conta própria. Uma maneira de fornecer essas instruções aos computadores é com um loop `for`.

Um `for` loop contém três expressões separadas por `;` parênteses:

1. uma *inicialização*: Inicia o loop e também pode ser usada para declarar a variável iteradora.
2. uma *condição de parada*: É a condição com a qual a variável do iterador é avaliada - se a condição for avaliada como verdadeiro, o bloco de código será executado e se for avaliada como falso, o código será interrompido.
3. uma *instrução de iteração*: É usada para atualizar a variável do iterador em cada loop.

```
// Exemplo
for (let counter = 0; counter < 4; counter++) {
  console.log(counter);
}
```

Neste exemplo, a saída seria a seguinte:

0  
1  
2  
3

## Loop reverso

E se quisermos que o `for` loop registre `3`, `2`, `1` e então `0`? Com modificações simples nas expressões, podemos fazer nosso loop rodar de trás para frente!

Para executar um loop reverso com `for`, devemos:

- Definir a variável iteradora para o valor desejado mais alto na expressão de inicialização.
- Defina a condição de parada para quando a variável do iterador for menor que o valor desejado.
- O iterador deve diminuir em intervalos após cada iteração.

## Loops através de arrays

A estrutura de repetição `for` são muito úteis para iterar estruturas de dados. Por exemplo, podemos usar um `for` loop para realizar a mesma operação em cada elemento de um array.

Os arrays podem conter, por exemplo, listas de dados, como nomes de clientes ou informações de produtos. Imagine que possuímos uma loja e queremos aumentar o preço de cada produto do nosso catálogo. Isso poderia ser muito código repetido, mas usando um `for` loop para iterar pelo array poderíamos realizar essa tarefa facilmente.

Para percorrer cada elemento de um array, um `for` loop deve usar a propriedade do array `.length` em sua condição.

```
const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];
for (let i = 0; i < animals.length; i++){
  console.log(animals[i]);
}
```

Este exemplo forneceria a seguinte saída:

Grizzly Bear  
Sloth  
Sea Lion

Com `for` loops, é mais fácil trabalharmos com elementos em arrays.

## Loops aninhados

Quando temos um loop rodando dentro de outro loop, chamamos isso de *loop aninhado*. Um uso para um `for` loop aninhado é comparar os elementos em dois arrays. Para cada iteração do loop `for` externo, o loop `for` interno será executado completamente.

Vejamos um exemplo de `for` loop aninhado:

```
const myArray = [6, 19, 20];
const yourArray = [19, 81, 2];
for (let i = 0; i < myArray.length; i++) {
  for (let j = 0; j < yourArray.length; j++) {
    if (myArray[i] === yourArray[j]) {
      console.log('Both arrays have the number: ' + yourArray[j]);
    }
  }
}
```

Vamos pensar no que está acontecendo no loop aninhado do nosso exemplo. Para cada elemento na matriz de loop externo `myArray`, o loop interno será executado em sua totalidade comparando o elemento atual da matriz externa, `myArray[i]` com cada elemento na matriz interna `yourArray[j]`. Quando encontra uma correspondência, ele imprime uma string no console.

## Loop while

vamos converter um `for` loop em `while` loop:

```
// Um for que retorna 1, 2, e 3
for (let counterOne = 1; counterOne < 4; counterOne++){
  console.log(counterOne);
}
```

```
// Um while que retorna 1, 2, e 3
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
}
```

- A

`counterTwo` variável é declarada antes do loop. Podemos acessá-lo dentro do nosso `while` loop, pois está no escopo global.

- Iniciamos nosso loop com a palavra-chave

`while` seguida por nossa condição de parada ou *condição de teste*. Isso será avaliado antes de cada rodada do loop. Enquanto a condição for avaliada como `true`, o bloco continuará em execução. Assim que for avaliado, `false` o loop irá parar.

## Do While

Em alguns casos, você deseja que um trecho de código seja executado pelo menos uma vez e, em seguida, faça um loop com base em uma condição específica após sua execução inicial. É aqui que `do...while` entra a afirmação.

Uma instrução `do...while` diz para executar uma tarefa uma vez e depois continuar a fazê-la até que uma condição especificada não seja mais atendida. A sintaxe de uma `do...while` instrução é semelhante a esta:

```
let countString = '';
let i = 0;

do {
  countString = countString + i;
  i++;
} while (i < 5);

console.log(countString);
```

Observe que o loop `while` e `do...while` são diferentes! Ao contrário do `while`, `do...while` será executado pelo menos uma vez, independentemente de a condição ser avaliada como `true`.

```
const firstMessage = 'Vou imprimir!';
const secondMessage = 'Não vou imprimir!';

// Um do while com uma condição de parada avaliada como falsa
do {
  console.log(firstMessage)
} while (true === false);

// Um loop while com uma condição de parada avaliada como falsa
while (true === false){
  console.log(secondMessage)
};
```

## A palavra-chave break

Em nosso código, quando queremos impedir que um loop continue a ser executado mesmo que a condição de parada original que escrevemos para nosso loop não tenha sido atendida, podemos usar a palavra-chave `break`.

O `break` permite que os programas “saiam” do loop a partir do bloco do loop.

```
for (let i = 0; i < 99; i++) {
  if (i > 2) {
    break;
  }
  console.log('Banana.');
```

```
console.log('Laranja, você está feliz por eu ter quebrado o c
```

`break` pode ser especialmente útil quando estamos percorrendo grandes estruturas de dados!

## Atividades de Aprendizagem

## **Exercício 1**

Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.

## **Exercício 2**

Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.

## **Exercício 3**

Faça um programa que leia 5 números e informe o maior número.

## **Exercício 4**

Faça um programa que leia 5 números e informe a soma e a média dos números.

## **Exercício 5**

Faça um programa que imprima na tela os números de 1 a 20, um abaixo do outro. Depois modifique o programa para que ele mostre os números um ao lado do outro.