

# AULA 1 – ARRAYS

## OBJETIVO DA AULA

Entender o que é, para que serve e como manipular um array em JavaScript.

## APRESENTAÇÃO

Nesta aula entraremos mais profundamente sobre o conceito de array, aprendendo o que é, para que serve e como criar um array em *JavaScript*. O array, muito utilizado na programação, nada mais é do que uma estrutura utilizada para armazenar e organizar valores em uma única variável. Esses valores, chamados de elementos, podem ser acessados rapidamente por suas posições numéricas, denominadas como *índices*. Vamos entender melhor sobre essa estrutura? Vem comigo!

### 1. ARRAY

Antes de definirmos array, é preciso que fique claro a diferença entre variáveis simples e compostas. Você já aprendeu o que é uma variável, lembra? Uma variável simples permite apenas o armazenamento de um único valor. Já a variável composta permite o armazenamento de vários elementos, sejam eles de um mesmo tipo de dados ou não, isso significa que podemos ter em um array um elemento inteiro (*int*), outro elemento *string* e outro elemento do tipo objeto. Comentamos acima que um array nada mais é do que uma estrutura utilizada para armazenar e organizar dados (elementos) em uma única variável. Logo, podemos concluir que um array é uma variável composta. Concorde?



#### DESTAQUE

Os arrays em *JavaScript* são dinâmicos. O que isso significa? Significa que podem crescer ou diminuir conforme a necessidade sem que seja necessário declarar um tamanho fixo para o array ao criá-lo ou ao realocá-lo, caso seu tamanho mude (FLANAGAN, 2013).

Para acessar os elementos que compõem o array utilizaremos a sua posição numérica, chamada de índice, que começa por zero. Lembrando que podemos trocar, acrescentar e remover elementos conforme a necessidade. E como podemos criar um array em *JavaScript*? “A maneira mais fácil de criar um array é com um array literal, que é simplesmente uma lista de elementos de array separados com vírgulas dentro de colchetes” (FLANAGAN, 2013). Veja:

- **var meuArray = [ ];**

//um array sem elementos.

- **var meuArray = [4, 5, 7, 10, 20];**

//um array com 5 elementos numéricos.

- **var meuArray = [2.5, true, "Ana"];**

//3 elementos de vários tipos.

Além disso, um array literal não necessariamente precisa ter valores constantes. Veja:

- **var x = 1024;**
- **var meuArray = [x, x+1, x+2 x+3].**

A outra maneira de criar array é utilizando o construtor **new Array()**. Essa construtora pode ser chamada de 3 formas diferentes (FLANAGAN, 2013):

- **var meuArray = new Array();**

//sem argumentos, ou seja, um array vazio. Equivalente ao array literal (var meuArray = [ ]).

- **var meuArray = new Array(10);**

//um array com comprimento especificado. Utilizado quando se sabe exatamente quantos elementos vão fazer parte do array, ou seja, este array terá 10 posições.

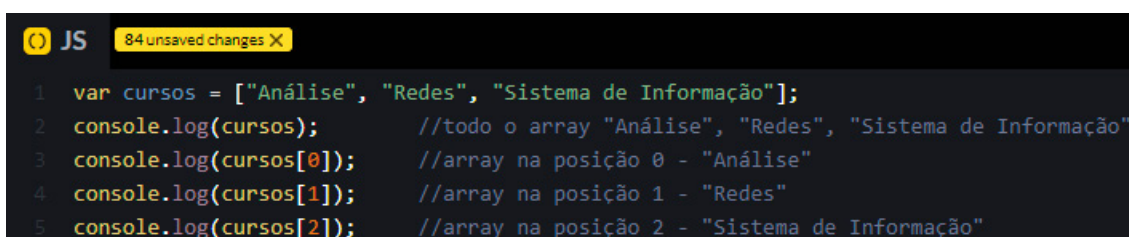
- **var meuArray = new Array(1, 2, 3, 4, 5, "teste").**

//neste caso os argumentos do construtor se tornam elementos do novo array.

Vale ressaltar que, apesar de termos diversas formas diferentes de criar um array, o tipo literal é o mais utilizado e recomendado. Por quê? Pelo simples fato de ser mais prático, de ter maior legibilidade e também por ser mais veloz em tempo de execução.

## 1.1. COMO ACESSAR OS ELEMENTOS DE UM ARRAY?

Para acessar os elementos de um array basta fazer a busca pelo seu índice. Veja:



```
JS 84 unsaved changes X
1 var cursos = ["Análise", "Redes", "Sistema de Informação"];
2 console.log(cursos);           //todo o array "Análise", "Redes", "Sistema de Informação"
3 console.log(cursos[0]);        //array na posição 0 - "Análise"
4 console.log(cursos[1]);        //array na posição 1 - "Redes"
5 console.log(cursos[2]);        //array na posição 2 - "Sistema de Informação"
```

Na *linha 2*, estamos exibindo na tela todos os elementos do array `cursos`. Nas *linhas 3-5*, estamos exibindo na tela o elemento específico de uma determinada posição. Fácil, não é mesmo? Também podemos percorrer o array utilizando o *for... of*, estrutura de repetição vista na unidade 2.



The screenshot shows a code editor with the following JavaScript code:

```
1 var cursos = ["Análise", "Redes", "Sistema de Informação"];
2 for (var elemento of cursos)
3   console.log(elemento);
```

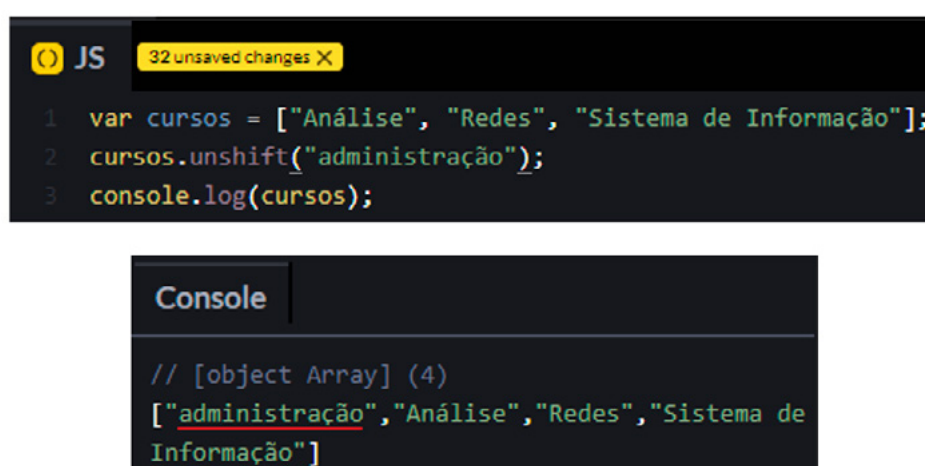
Below the code, the console output is displayed:

```
Console
"Análise"
"Redes"
"Sistema de Informação"
```

Como dito anteriormente, o array é uma estrutura não-tipada e dinâmica, ou seja, um elemento pode ser de qualquer tipo e um array pode crescer ou diminuir segundo a necessidade. Sendo assim, vamos conhecer alguns métodos que facilitam bastante o dia a dia dos desenvolvedores.

## 1.2. COMO ADICIONAR ELEMENTOS EM UM ARRAY?

- **unshift()** – adiciona um ou mais elementos no início do array. Veja:



The screenshot shows a code editor with the following JavaScript code:

```
1 var cursos = ["Análise", "Redes", "Sistema de Informação"];
2 cursos.unshift("administração");
3 console.log(cursos);
```

Below the code, the console output is displayed:

```
Console
// [object Array] (4)
["administração", "Análise", "Redes", "Sistema de Informação"]
```

- **push()** – adiciona um ou mais elementos no final do array. Veja:

```
JS 29 unsaved changes X
1 var cursos = ["Análise", "Redes", "Sistema de Informação"];
2 cursos.push("administração");
3 console.log(cursos);
```

```
Console
// [object Array] (4)
["Análise","Redes","Sistema de
Informação","administração"]
```

- **splice()** – altera o conteúdo do array, adicionando novos elementos enquanto remove elementos antigos. Veja:

```
JS 84 unsaved changes X
1 let mes = ['Jan', 'Mar', 'Abril', 'Jun'];
2 mes.splice(1, 0, 'Fev'); // Inserção de FEV no índice 1
3 console.log(mes);
4 //Resultado: ['Jan', 'Fev', Mar', 'Abril', 'Jun']
5
6 mes.splice(4, 1, 'Maio'); // substituí o elemento do índice 4 por MAIO
7 console.log(mes);
8 //Resultado: ['Jan', 'Fev', Mar', 'Abril', 'Maio']
```

Na primeira parte do exemplo, estamos usando o `splice(1, 0, 'Fev')` para inserir o mês de “Fev” no índice 1, o segundo argumento foi 0 porque neste caso não queremos remover nenhum elemento do array. Na segunda parte do exemplo, estamos usando o `splice(4, 1, 'Maio')` para remover o elemento do índice 4 (Jun) e substituir por “Maio”, para isso, tivemos que inserir o número 1 no segundo argumento para indicar que apenas 1 elemento será removido.

### 1.3. COMO REMOVER ELEMENTOS DE UM ARRAY?

- **shift()** – remove um ou mais elementos no início do array. Veja:

```

1 var cursos = ["Administração", "Análise", "Redes", "Sistema de Informação"];
2 cursos.shift();
3 console.log(cursos);

```

**Console**

```

// [object Array] (3)
["Análise","Redes","Sistema de Informação"]

```

- **pop()** – remove um ou mais elementos no final do array. Veja:

```

1 var cursos = ["Análise", "Redes", "Sistema de Informação", "Administração"];
2 cursos.pop();
3 console.log(cursos);

```

**Console**

```

// [object Array] (3)
["Análise","Redes","Sistema de Informação"]

```

## 1.4. COMO ORDENAR UM ARRAY NUMÉRICO?

Para ordenar um array numérico basta utilizarmos o método `sort()`. Veja:

```

1 let idades = [38, 59, 28, 64, 60];
2 idades.sort();
3 console.log(idades);

```

**Console**

```

// [object Array] (5)
[28,38,59,60,64]

```

## CONSIDERAÇÕES FINAIS

Nesta aula você teve a chance de entender melhor o que é e para que serve um array. Podemos dizer que o conceito de array é um dos mais importantes na programação, pois traz muita praticidade no desenvolvimento de aplicações mais robustas e complexas. Lembrando que um array em *JavaScript* não possui tamanho nem tipo fixos, com isso, podemos acrescentar, remover e trocar os elementos de posição livremente. Aprendemos como criar um array,

O conteúdo deste livro eletrônico é licenciado para GLEITON - 08303020692, vedada, por quaisquer meios e a qualquer título, a sua reprodução, cópia, divulgação ou distribuição, sujeitando-se aos infratores à responsabilização civil e criminal.

conhecemos alguns métodos usados para incluir, alterar, excluir e ordenar elementos de um array. Legal, né? Não esqueça de acessar o material complementar, ele é muito importante para arrematar os conhecimentos adquiridos aqui. Na próxima aula vamos falar sobre o arrays multidimensionais. Te espero!

## MATERIAIS COMPLEMENTARES

Javascript Array – aprenda o que é, como criar e usar:

<https://blog.betrybe.com/javascript/javascript-array/>

Variáveis Compostas – Curso JavaScript:

<https://youtu.be/XdkW62tkAgU>

## REFERÊNCIAS

FLANAGAN, David. *JavaScript: O guia definitivo*. Porto Alegre, RS: Bookman, 2013.

# AULA 2 – ARRAYS MULTIDIMENSIONAIS

## OBJETIVO DA AULA

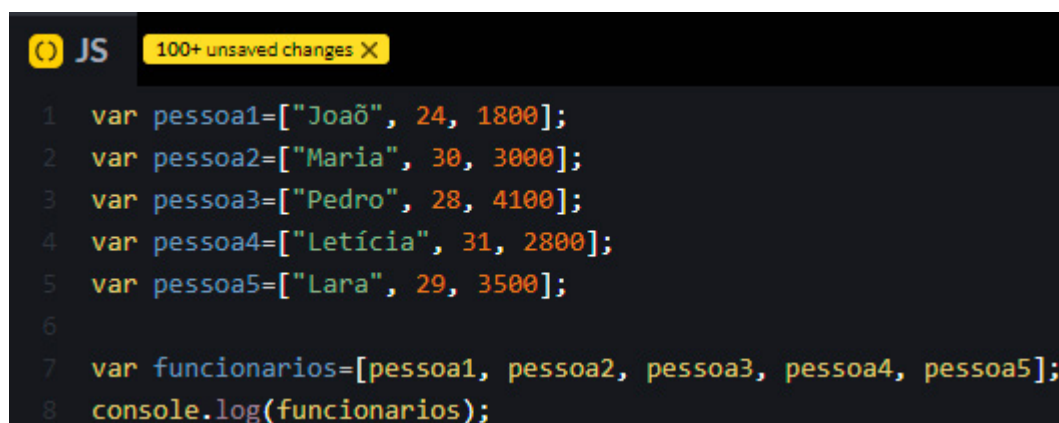
Entender o que é, para que serve e como manipular um array multidimensional em JavaScript.

## APRESENTAÇÃO

Na aula passada aprendemos que um array nada mais é do que uma estrutura utilizada para armazenar e organizar valores em uma única variável. Nesta aula abordaremos o conceito de array multidimensional, ou seja, um array com múltiplas dimensões. Calma! Não se assuste! Um array multidimensional nada mais é do que um array que armazena outro array. É isso mesmo! Em outras linguagens de programação, um array de array é chamado de matriz. Não podemos esquecer que um array no *JavaScript* é um tipo especial de objeto que trabalha apenas com índices numéricos. Vamos juntos aprender como criar um array de array e como adicionar e remover elementos do mesmo.

## 1. ARRAYS MULTIDIMENSIONAIS

“*JavaScript* não suporta arrays multidimensionais de verdade, mas é possível ter algo parecido, como array de array” (FLANAGAN, 2013). Não é nada muito diferente do que você já sabe, por exemplo, para acessar um elemento de um array simples (unidimensional) você utiliza um único colchete [ ] (meuArray[x]) e para acessar um elemento de um array de array você deve utilizar o colchete [ ] duas vezes (meuArray[x][y]). Assim como no array simples, a forma mais fácil de criar um array de array é usar a notação literal de array. Vamos ver duas formas diferentes de criar um array com múltiplas dimensões. A primeira forma é:



```

JS 100+ unsaved changes X
1  var pessoa1=["João", 24, 1800];
2  var pessoa2=["Maria", 30, 3000];
3  var pessoa3=["Pedro", 28, 4100];
4  var pessoa4=["Letícia", 31, 2800];
5  var pessoa5=["Lara", 29, 3500];
6
7  var funcionarios=[pessoa1, pessoa2, pessoa3, pessoa4, pessoa5];
8  console.log(funcionarios);
  
```



```

Console

// [object Array] (5)
* [// [object Array] (3)
  ["Joaõ",24,1800],// [object Array] (3)
  ["Maria",30,3000],// [object Array] (3)
  ["Pedro",28,4100],// [object Array] (3)
  ["Letícia",31,2800],// [object Array] (3)
  ["Lara",29,3500]]

```

Criar um array (*linha 7*) que vai receber todos os arrays unidimensionais criados anteriormente (*linhas 1-5*), ou seja, o array funcionário agrupa todos os arrays pessoa. Agora vamos ver a segunda forma:

```

JS 100+ unsaved changes X
1 var funcionarios=[
2   ["Joaõ", 24, 1800],
3   ["Maria", 30, 3000],
4   ["Pedro", 28, 4100],
5   ["Letícia", 31, 2800],
6   ["Lara", 29, 3500],
7 ];
8 console.log(funcionarios);
9
Console
// [object Array] (5)
* [// [object Array] (3)
  ["Joaõ",24,1800],// [object Array] (3)
  ["Maria",30,3000],// [object Array] (3)
  ["Pedro",28,4100],// [object Array] (3)
  ["Letícia",31,2800],// [object Array] (3)
  ["Lara",29,3500]]

```

Aqui temos o array funcionários funcionando como uma array multidimensional e produzindo o mesmo resultado do exemplo anterior. Este resultado pode ser mostrado em um formato de tabela, para isso, basta usar o método `console.table()` ao invés do `console.log()`. Veja:

```

1 var funcionarios=[
2   ["Joaõ", 24, 1800],
3   ["Maria", 30, 3000],
4   ["Pedro", 28, 4100],
5   ["Letícia", 31, 2800],
6   ["Lara", 29, 3500],
7 ];
8 console.table(funcionarios);

```

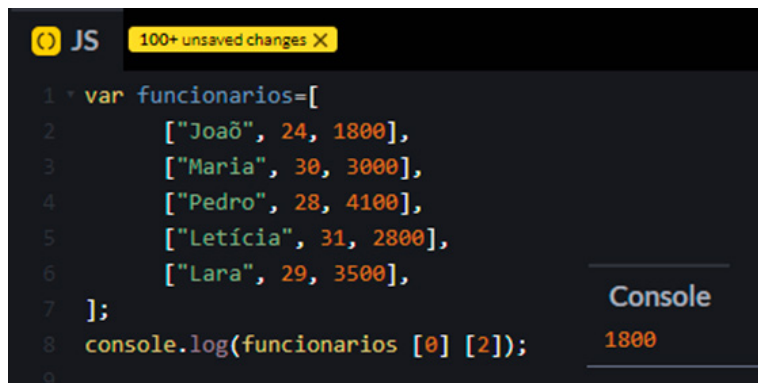
(index)	0	1	2
0	'Joaõ'	24	1800
1	'Maria'	30	3000
2	'Pedro'	28	4100
3	'Letícia'	31	2800
4	'Lara'	29	3500

Esta visualização conta os índices que representam cada um dos elementos presentes no array de arrays.



## 1.1. COMO ACESSAR UM ARRAY DE ARRAYS?

Podemos acessar os elementos de array de arrays usando a notação simples baseada em índice ou usando uma estrutura de repetição, assim como no array unidimensional. Veja:



```

1 * var funcionarios=[
2     ["João", 24, 1800],
3     ["Maria", 30, 3000],
4     ["Pedro", 28, 4100],
5     ["Letícia", 31, 2800],
6     ["Lara", 29, 3500],
7 ];
8 console.log(funcionarios [0] [2]);
9

```

Console  
1800

Neste exemplo, estamos acessando o salário do João (funcionários [0] [2]), onde **[0]** seleciona a primeira linha e **[2]** seleciona o terceiro elemento dessa primeira linha que é 1800.

Veja agora como percorrer o vetor usando uma estrutura de repetição:



```

1 * var funcionarios=[
2     ["João", 24, 1800],
3     ["Maria", 30, 3000],
4     ["Pedro", 28, 4100],
5     ["Letícia", 31, 2800],
6     ["Lara", 29, 3500],
7 ];
8 * for(var i=0; i<funcionarios.length; i++){
9     for(var j=0; j<funcionarios[i].length; j++){
10         console.table(funcionarios[i][j]);
11     }
12 }

```

Console	
"João"	
24	
1800	"Letícia"
"Maria"	31
30	2800
3000	"Lara"
"Pedro"	29
28	3500
4100	

Esta versão é um pouco mais complexa, pois estamos fazendo uso de uma estrutura de repetição aninhada ou encadeada, isto significa que para cada rodada do primeiro *for*, o segundo é executado por completo. O primeiro *for* percorre os elementos do array externo e o segundo *for* percorre os elementos internos do array interno. Precisamos de duas estruturas de repetição aninhadas porque um array de arrays possui múltiplas linhas. Funciona mais ou menos assim: o primeiro *for* chega na primeira linha e o segundo passa por todos os elementos da linha 1, em seguida o primeiro *for* passa para a segunda linha e o segundo passa por todos os elementos da segunda linha e assim sucessivamente.



### PRA PRATICAR

Os métodos: `push()`, `pop()`, `shift()`, `unshift()` e `splice()` também podem ser utilizados para manipular arrays multidimensionais. Sendo assim, deixo como desafio para você tentar incluir e excluir elementos em um array de arrays.

## CONSIDERAÇÕES FINAIS

Nesta aula você aprendeu que o *JavaScript* não fornece o array multidimensional nativamente, para contornar esta situação temos que criar um array de arrays. Certamente, isso não é um problema para você, não é mesmo? Na aula anterior aprendemos como criar e manipular arrays unidimensionais. Então, aqui é quase a mesma coisa, a diferença está na quantidade de índices que cada elemento da nossa estrutura terá como identificado, por exemplo, `meuArray [5][5]` tem duas dimensões e cada dimensão possui 5 elementos. Logo, temos uma estrutura 5x5, capaz de armazenar 25 elementos distintos. Na próxima aula falaremos sobre funções. Te espero!

## MATERIAIS COMPLEMENTARES

Curso JavaScript|Array Multidimensional – Matrizes|Coffee Tag:

<https://youtu.be/F1SIGxIQ-90>

Curso JavaScript|Adicionar um elemento a uma matriz Multidimensional|Coffee Tag:

<https://youtu.be/8Yz2csLjZAk>

Curso JavaScript|Deletar um valor da Array ou Matriz Multidimensional|Coffee Tag:

[https://youtu.be/P8RIL\\_IlkOE](https://youtu.be/P8RIL_IlkOE)

## REFERÊNCIAS

FLANAGAN, David. *JavaScript: O guia definitivo*. Porto Alegre, RS: Bookman, 2013.

# AULA 3 – FUNÇÕES

## OBJETIVO DA AULA

Encapsular um código que poderá ser invocado/chamado por qualquer outro trecho do programa.

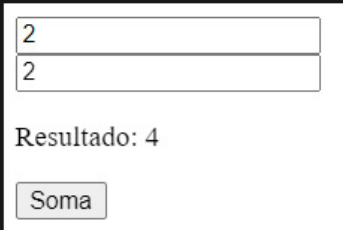
## APRESENTAÇÃO

Para complementar ainda mais o nosso conhecimento, nesta aula vamos aprender mais profundamente o conceito de função. Uma função nada mais é do que um bloco de instruções que executa uma determinada tarefa ao ser “chamada” ou “invocada”. Nós já tivemos contato com o conceito de função na *unidade 1* (“criando soluções”), lembra?

```

1  function Soma()
2  {
3      var num1 = parseInt(document.getElementById("n1").value);
4      var num2 = parseInt(document.getElementById("n2").value);
5      var soma = num1 + num2;
6      document.getElementById("res").innerHTML = "Resultado: " + soma;
7  }

```



Fizemos juntos este exemplo de como somar dois números digitados pelo usuário, ou seja, definimos a nossa função chamada `Soma()` que é executada quando o usuário clica no botão SOMA.

## 1. FUNÇÕES

Assim como o array, o uso de funções é algo muito comum nas linguagens de programação. Sabemos que cada linguagem tem as suas particularidades e maneiras específicas de como lidar com as funções. “Uma função é um bloco de código definido uma vez, mas que pode ser executado ou chamado qualquer número de vezes” (FLANAGAN, 2013). Em *JavaScript*, as funções são objetos e podem ser manipuladas pelos programas. Podemos defini-las de 5 formas diferentes (CASTIGLIONI, 2022):

- *Functions declaration* (Função de declaração);

Livro Eletrônico

- *Functions expression* (Função de expressão);
- *Arrow functions* (Função de flecha);
- *Functions constructor* (Função construtora);
- *Generator functions* (Função geradora).
- **Função de declaração (*Functions declaration*)**: essa é a forma mais comum, nela basicamente usamos a palavra-chave *function* seguida pelo nome da função (obrigatório) e os parênteses (), que representam os parâmetros (opcional) da função. Veja a sintaxe de uma função:

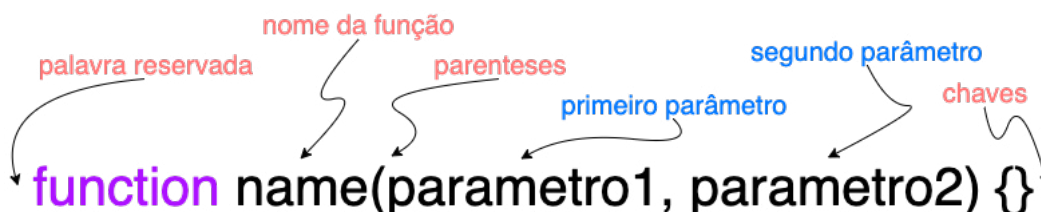
FIGURA 1 | Sintaxe de função de declaração sem parâmetros



Fonte: <https://blog.matheuscastiglioni.com.br/definindo-funcoes-em-javascript/>

Esta é a sintaxe de uma função sem parâmetros. Agora veja a sintaxe de uma função com parâmetros, repare que os parâmetros são separados por vírgula.

FIGURA 2 | Sintaxe de função de declaração com parâmetros



Fonte: <https://blog.matheuscastiglioni.com.br/definindo-funcoes-em-javascript/>

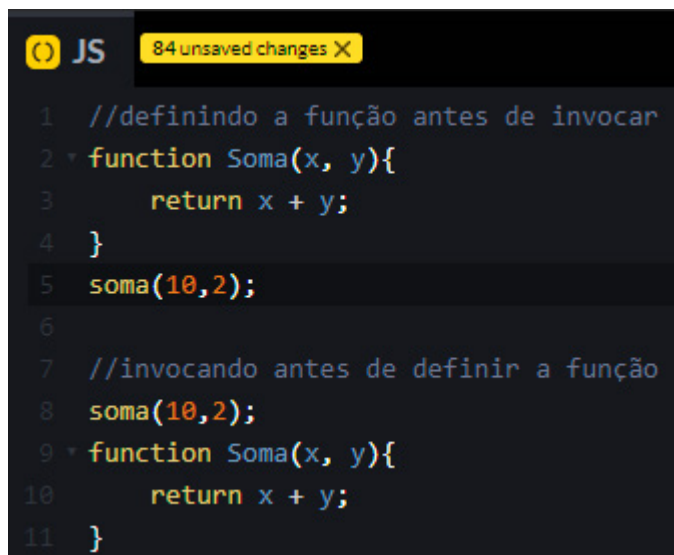
Vamos ver agora outra versão para a nossa função Soma():

```
JS 62 unsaved changes X
1 function Soma(x, y){
2   return x + y;
3 }
```

Neste exemplo definimos uma função que recebeu 2 argumentos (x, y) como parâmetro e retornou a soma dos mesmos.

## DESTAQUE

Vale ressaltar que, “funções de declaração” podem ser “chamadas” ou “invocadas” antes ou após serem definidas. Como assim? Veja:



```

1 //definindo a função antes de invocar
2 function Soma(x, y){
3     return x + y;
4 }
5 soma(10,2);
6
7 //invocando antes de definir a função
8 soma(10,2);
9 function Soma(x, y){
10     return x + y;
11 }
    
```

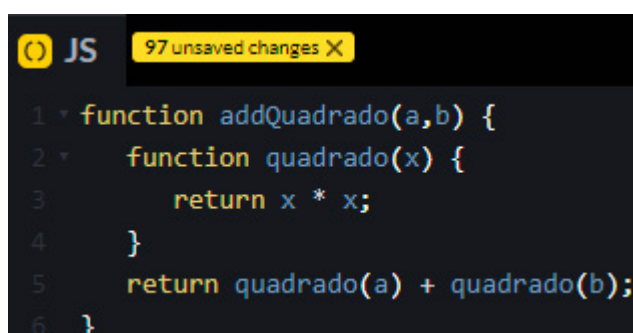
Isso é chamado de *function hoisting*.

Durante a fase de criação da memória, a *engine* JavaScript reconhece uma declaração de função pela palavra-chave *function* — ou seja, a *engine* JavaScript disponibiliza a função colocando-a na memória antes de prosseguir. Por isso, ela está disponível aparentemente antes da definição da mesma quando se lê o código de cima para baixo (CASTIGLIONI, 2022).

Lembra que podemos aninhar estruturas condicionais e de repetição?

## DESTAQUE

Então, “as definições de função JavaScript também podem ser aninhadas dentro de outras funções e têm acesso a qualquer variável que esteja no escopo onde são definidas” (FLANAGAN, 2013). Veja:

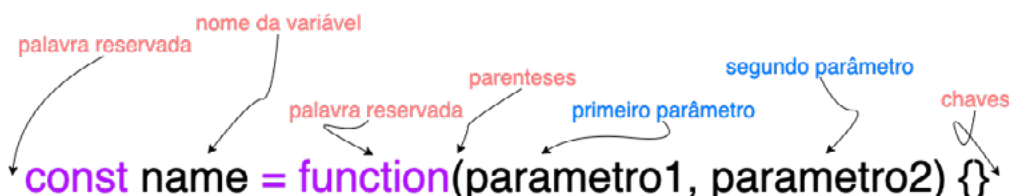


```

1 function addQuadrado(a,b) {
2     function quadrado(x) {
3         return x * x;
4     }
5     return quadrado(a) + quadrado(b);
6 }
    
```

- **Função de expressão (*Functions expression*):** a função de expressão é muito parecida com a função de declaração, a diferença é que ela pode ser armazenada em uma atribuição de variável e seu nome é opcional, ou seja, uma vez atribuída a uma variável, ela pode ser “chamada” pelo próprio nome da variável. Veja:

FIGURA 3 | **Sintaxe de função de expressão com parâmetros**



Fonte: <https://blog.matheuscastiglioni.com.br/definindo-funcoes-em-javascript/>

Vamos ver agora como ficaria a definição da nossa função Soma():

Neste exemplo não definimos o nome da função (Soma), mas, sim, o nome da variável (somatório) que irá referenciar a mesma. Bom, você deve estar se perguntando: “Qual a vantagem de atribuir uma função a uma variável?” Simples! Ao atribuir uma função a uma variável:

(...) podemos definir a função exatamente onde ela precisa ser chamada, ou seja, definimos a função apenas onde precisamos dela, isso em alguns momentos pode tornar nosso código mais simples de entender (CASTIGLIONI, 2022).

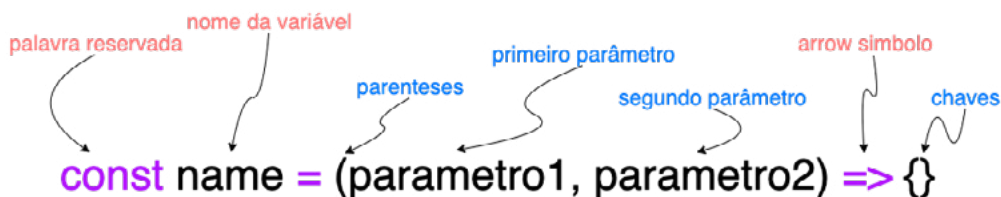
### DESTAQUE

Funções de expressão, diferentemente das funções de declaração, não pode ser chamadas antes de serem definidas no código.

- **Função de Flecha (*Arrow functions*):** A função de flecha é um conceito relativamente novo e serve para simplificar as definições de função. Nela temos um novo operador =>, porém não precisamos usar a palavras reservadas *function*, se o corpo da função tiver apenas uma linha não é necessário o uso de {} e nem da palavra reservada *return*, se a função tem apenas um parâmetro, os parênteses são opcionais, enfim, a ideia aqui é simplificar as funções de declaração (*Functions declaration*) e de expressão (*Functions expression*). Veja como seria a sua sintaxe:



FIGURA 4 | Sintaxe de função de flecha com parâmetros



Fonte: <https://blog.matheuscastiglioni.com.br/definindo-funcoes-em-javascript/>

Agora veja como ficaria no nosso exemplo da função para somar 2 números, levando em consideração todas aquelas vantagens citadas acima:

```
JS 100+ unsaved changes X
1 let somatorio = (x, y) => x + y;
2 //Aqui estamos chamando a função somatorio e o resultado obtido é 7
3 console.log(somatorio(2,5));
```

Viu com é mais simples? Em uma única linha (*linha 1*) definimos uma função *Arrow function* com que recebeu dois argumentos e retornou a nossa dos mesmos. Não foi necessário o uso das palavras reservadas *function* e *return* e também não usamos *{}*. Na *linha 3* estamos chamando a função, passamos os valores 2 e 5 como parâmetro e obtivemos o valor 7 como resposta, ou seja, ela está somando direitinho.

- **Função Construtora (*Functions constructor*):** A diferença entre ela e as demais é como ela é “chamada” ou “invocada”. As funções construtoras precisam ser invocadas com a palavra reservada *new*. A vantagem é que com ela podemos definir a função e o corpo da função simultaneamente. Veja o nosso exemplo de somar dois números:

```
JS 100+ unsaved changes X
1 let somatorio = new Function('x', 'y', 'return x + y');
2 console.log(somatorio(2, 5)); //imprime 7
```

Repare que passamos três argumentos, os dois primeiros serão os parâmetros (x, y) da função que está sendo criada e o último (return x + y) é a definição do corpo da função. Agora veja um exemplo de utilização da função construtora para criar um objeto (já vimos como criar um objeto na Unidade 2):

```
JS 100+ unsaved changes X
1 function Pessoa(nome, sobrenome){
2   this.nome=nome;
3   this.sobrenome=sobrenome;
4 }
5 const teste = new Pessoa('Natália', 'Oliveira');
6 console.log(teste);

// [object Object]
{
  "nome": "Natália",
  "sobrenome": "Oliveira"
}
```

Neste exemplo estamos criando o objeto **Pessoa** com as propriedades nome e sobrenome. A palavra `new` (linha 5) foi utilizada para chamar a função `Pessoa`, o *JavaScript*, no que lhe concerne, cria automaticamente um objeto para nós e o mesmo pode ser referenciado através do `this`. Quando fazemos `this.nome=nome` e `this.sobrenome=sobrenome`, estamos adicionando as propriedades nome e sobrenome para o objeto `Pessoa`, onde os valores são informados no parâmetro da função ('Natália' 'Oliveira').

**Obs.:** | Normalmente o nome das funções construtoras começam com maiúsculo.

- **Função Geradora (*Generator functions*):** a função geradora também é um conceito novo, cujo objetivo é retornar uma sequência de valores. Toda vez que a função é “chamada” ela retorna um valor até que o último seja retornado. Confuso, né? Vamos entender melhor... Nós não temos controle do que será executado em uma função, concorda? Em todos os tipos que vimos até o momento, a função sempre é executada por completo. Na função geradora é diferente; aqui temos o total controle da situação, ou seja, podemos interrompê-la durante a sua invocação e posteriormente podemos dar continuidade em sua execução. A diferença visual entre ela e as demais é que na função geradora utilizamos o `*` logo depois da palavra reservada `function`. Veja:

```

1 * function* testandoGeradora() {
2     yield 1;
3     yield 2;
4     yield 3;
5 }
6 const funcaoGeradora = testandoGeradora();
7 console.log(funcaoGeradora.next());
8 console.log(funcaoGeradora.next());
9 console.log(funcaoGeradora.next());
10 console.log(funcaoGeradora.next());
11

```

The console output shows four objects returned by the generator function's `next()` method:

```

// [object Object]
{
  "value": 1,
  "done": false
}

// [object Object]
{
  "value": 2,
  "done": false
}

// [object Object]
{
  "done": true
}

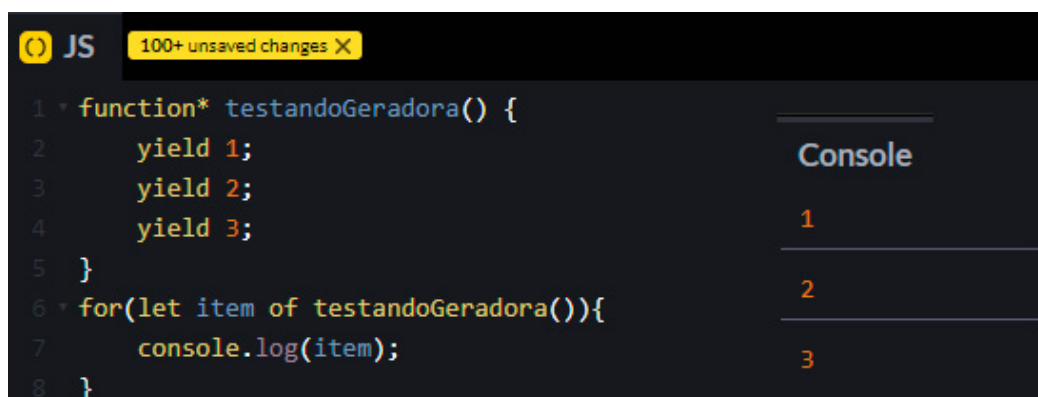
// [object Object]
{
  "value": 3,
  "done": false
}

```

Vale ressaltar que uma função geradora quando invocada sempre retorna um **objeto iterador**. O que isso significa? Conforme a página oficial da MDN (2022),

(...) em JavaScript um **iterator** é um objeto que oferece o método `next()`, o qual retorna o próximo item da sequência. Este método retorna um objeto com duas propriedades: **done** e **value**.

Com isso, ficou mais fácil de entendermos o trecho de código acima. A palavra reservada `yield` (linhas 2 – 4) é como se fosse um ponto de interrupção, ou seja, indica onde a função deve ir parando sua execução. Cada vez que “chamamos” a função usando o `next()` (linhas 7-9), ela retorna um `yield`. A propriedade `value` retorna o valor informado para cada `yield` e a propriedade `done` indica se o iterator percorreu todos os `yields`, quando obtemos o valor `true` significa que a iteração terminou. Também podemos percorrer o resultado de uma função geradora utilizando uma estrutura de repetição. Veja:



```

1 * function* testandoGeradora() {
2     yield 1;
3     yield 2;
4     yield 3;
5 }
6 for(let item of testandoGeradora()){
7     console.log(item);
8 }
  
```

Console

1

2

3

Neste caso, não utilizamos o método `next()`, logo não é necessário se preocupar em verificar se tem `value` e se o `done` não está `true`, pois a própria estrutura de repetição, no nosso caso o `for`, faz tudo isso para gente.

## CONSIDERAÇÕES FINAIS

E aí pessoal? Gostaram? Nesta aula vimos que trabalhar com funções é algo relativamente simples: basta declararmos a função e seu escopo e depois chamá-la para que seu código seja executado. Porém, vimos também que existem diversas formas diferentes de fazer isso, cada uma delas tem sua particularidade. Qual delas é a preferida de vocês? Aproveitem para aprender ainda mais a diferença entre elas praticando e testando todos os códigos que vimos aqui. Espero por vocês na próxima aula!

## MATERIAIS COMPLEMENTARES

Funções:

<https://youtu.be/mc3TKp2Xzhl>

7 maneiras para criar funções com Javascript:

<https://youtu.be/dla0lYCwbYk>

## REFERÊNCIAS

CASTIGLIONI; Matheus. *Definindo funções em Javascript*, 2022. Disponível em: <https://blog.matheuscastiglioni.com.br/definindo-funcoes-em-javascript/>. Acesso em 17 de nov. de 2022.

MDN. *Iteradores e geradores*, 2022. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Functions>>. Acesso em 19 de nov. de 2022.

# AULA 4 – RECURSIVIDADE

## OBJETIVO DA AULA

Aplicar o conceito de recursão em *JavaScript*.

## APRESENTAÇÃO

Nesta aula vamos falar de recursividade; este novo conceito tem tudo a ver com a aula de funções. Antes mesmo de falar de recursividade em JavaScript, vamos entender o conceito de recursão:

Recursão é um método de resolução de problemas que envolve quebrar um problema em subproblemas menores e menores até chegar a um problema pequeno o suficiente para que ele possa ser resolvido trivialmente. Normalmente recursão envolve uma função que chama a si mesma. Embora possa não parecer muito, a recursão nos permite escrever soluções elegantes para problemas que, de outra forma, podem ser muito difíceis de programar (MILER, 2014).

Vem comigo para entender melhor este conceito que parece tão complexo.



### LINK

Saiba mais sobre o conceito de recursão no nosso blog: <https://blog.grancursosonline.com.br/recursividade-na-logica-de-programacao/>. Acesso em: 05/01/2023.



## 1. RECURSIVIDADE

Segundo a Wikipedia (2021), recursividade na ciência da computação nada mais é do que “a definição de uma sub-rotina (função ou método) que pode invocar a si mesma”, por exemplo, uma função chamando outra função. Basicamente seria assim:

```
JS 100+ unsaved changes X
1 function recursividade(){
2   //instruções
3   recursividade();
4   //instruções
5 }
```

Livro Eletrônico

A recursividade pode substituir as estruturas de repetição, sua utilização depende do problema a ser resolvido, pois em alguns casos é mais fácil usar a estrutura de repetição e em outros casos é mais fácil usar a recursividade. É importante lembrar que uma função recursiva precisar ter uma condição que interrompa a invocação dela mesma, caso contrário entrará num *loop* infinito. Ficaria assim:

```
JS 100+ unsaved changes X
1 function recursividade(){
2   if (condition){
3     //para de se chamar
4   }
5   else{
6     recursividade();
7   }
```

Um exemplo clássico de recursividade é criar uma função para calcular o fatorial (multiplicação de todos os números de uma sequência de 1 até o limite) de um número. Vamos fazer juntos? Antes de mais nada vamos relembrar como calcular o fatorial de um número, no nosso caso será o número 5:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

ou

$$5 \times 4 = 20$$

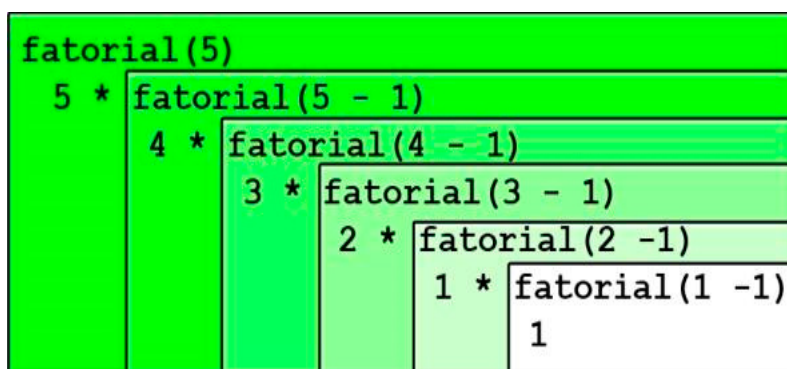
$$20 \times 3 = 60$$

$$60 \times 2 = 120$$

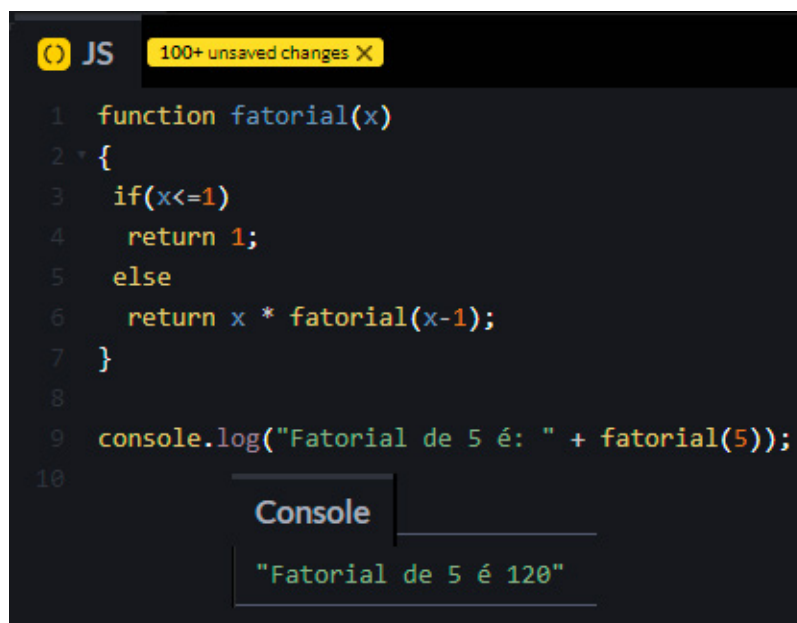
$$120 \times 1 = 120$$

Para criar uma função recursiva para resolver este caso devemos multiplicar o número 5 pelo valor anterior (4). No próximo passo a mesma coisa, multiplicar o resultado obtido (20) pelo valor anterior (3) e assim por diante até que o valor seja um para indicar o fim do processo. Veja:

FIGURA 1 | Exemplo de função recursiva



Agora vamos colocar a mão na massa para ver como ficará o código em JavaScript:



```

1  function fatorial(x)
2  * {
3    if(x<=1)
4      return 1;
5    else
6      return x * fatorial(x-1);
7  }
8
9  console.log("Fatorial de 5 é: " + fatorial(5));
10

```

Console

"Fatorial de 5 é 120"

Neste exemplo criamos uma função chamada *fatorial* que está recebendo um valor como parâmetro "x" (*linha 1*). Em seguida, verificamos se o valor x é menor ou igual a 1 (*linha 3*), caso o resultado seja verdadeiro o valor 1 é retornado para função, caso o resultado seja falso, inicia-se o processo de recursividade. Na *linha 6*, o retorno da função *fatorial* é a multiplicação do valor passado como parâmetro "x" pelo retorno da função *fatorial* com o valor de "x-1" (`return x * fatorial(x-1)`), como observamos no exemplo acima, a primeira iteração (ciclo) é  $5 * 4$  cujo resultado é 20, logo, 20 será multiplicado por  $x - 1$  e assim sucessivamente.



### PRA PRATICAR

Como seria a função *fatorial* sem a recursividade? Coloque a mão na massa e tente fazer sozinho uma função que calcule o fatorial utilizando iteração e não recursão.



Agora vamos ver um mesmo exemplo (contagem regressiva) utilizando a estrutura de repetição *for* e a recursividade. Com o *for* ficaria assim:

The screenshot shows a code editor with a JavaScript file. The code is a `for` loop that starts at 10 and decrements until it reaches 1, logging each value to the console. The console output on the right shows the sequence of numbers from 10 down to 1.

```

1 for(var cont=10; cont>=1; cont--)
2   console.log(cont);

```

Console output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Utilizando a recursividade ficaria assim:

The screenshot shows a code editor with a JavaScript file. The code defines a recursive function `regressiva` that takes a current count and a target value. It logs the current count and calls itself with the decremented count until it reaches the target. The console output on the right shows the sequence of numbers from 10 down to 1.

```

1 function regressiva(cont,num){
2   console.log(cont);
3   if(num < cont){
4     regressiva(--cont,num);
5   }
6 }
7
8 regressiva(10,1);

```

Console output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Neste exemplo criamos uma função chamada **regressiva** que recebeu dois parâmetros de entrada: *cont* que representa o valor inicial e *num* que representa o valor final da contagem regressiva (*linha 1*). Dentro da função, a primeira linha (*linha 2*) mostra no console o primeiro valor da variável *cont* que neste caso é 10. Em seguida (*linha 3*) temos a estrutura condicional *if* que verificará a condição estabelecida para a nova chamada da função **regressiva**. Caso a condição seja verdadeira, a função é chamada novamente (*linha 4*), caso contrário, tudo é encerrado. Repare que a variável *cont* está sendo decrementada como pré-incremento (`--cont`)

e não como pós-incremento (`cont++`), pois neste caso é necessário que a variável `cont` já esteja decrementada. Para dar início ao processo de recursividade precisamos chamar a função (linha 8) regressiva (10, 1) e passar os valores inicial (10) e final (1) da contagem regressiva.

Resumindo, iniciamos a chamada da função com os valores `cont=10` e `num=1`. A função então verifica SE `num` é menor que `cont`, sendo menor este teste retorna `true` e a função **regressiva** é chamada novamente, porém, com o valor de `cont` decrementado (`--`) em um. Então chama a função **regressiva** com os valores `cont=9` e `num=1`, assim o processo é repetido, até que `cont` tenha o valor 1, neste momento o teste irá retornar `false` e não chamará a função novamente, saindo assim da recursividade.

## CONSIDERAÇÕES FINAIS

Nesta aula falamos sobre um assunto que espanta muita gente que é a Recursividade. Aprendemos que uma função recursiva é aquela que chama a si mesma até que seja interrompida. No início pode parecer um tanto quanto bizarro, mas tenho certeza de que agora você já está tirando de letra. Lembrando que a prática é fundamental para absorção do conceito. Vimos que a recursividade torna o nosso código mais elegante e tem o poder de substituir uma estrutura de repetição, porém é preciso tomar cuidado para não colocar o seu código em *loop* infinito.

## MATERIAIS COMPLEMENTARES

Recursividade/Dicionário de Programação:

<https://youtu.be/NKymAD4pJZI>

Lógica de Programação – Recursividade:

<https://youtu.be/M7c-m2xN9FQ>

## REFERÊNCIAS

MILLER, Brad et al. *How to think like a computer scientist: Interactive edition*. Runestone, 2014.

WIKIPEDIA. *Recursividade (ciência da computação)*, 2021. Disponível em: < [https://pt.wikipedia.org/wiki/Recursividade\\_\(ci%C3%A2ncia\\_da\\_computa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Recursividade_(ci%C3%A2ncia_da_computa%C3%A7%C3%A3o)) >. Acesso em 19 de nov. de 2022.

# AULA 5 – IMERSÃO JAVASCRIPT – COLEÇÕES E FUNÇÕES

## OBJETIVO DA AULA

Praticar os conceitos vistos até o momento.

## APRESENTAÇÃO

Nesta aula iremos colocar a mão na massa. Isso mesmo! Vamos fazer exemplos práticos juntos para testar os nossos conhecimentos em relação aos conceitos aprendidos nas aulas anteriores:

- Arrays;
- Arrays multifuncionais;
- Funções;
- Recursividade.

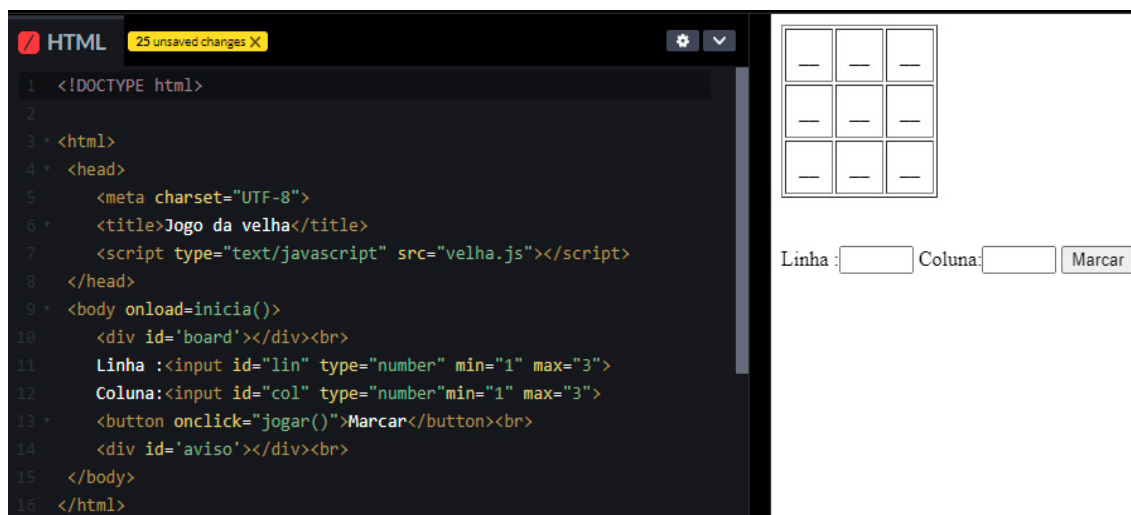
Basicamente, vamos fazer um exercício que envolva vários dos conceitos aprendidos aqui, assim teremos ideia de como esses assuntos trabalham em conjunto. A prática é muito importante nesta disciplina. Preparados? Vamos lá!

## 1. EXEMPLO 1 – ARRAY + ARRAY MULTIDIMENSIONAL + FUNÇÃO

Neste exemplo vamos aprender a construir um jogo da velha usando Javascript. Este exercício faz parte de um livro de De Matos (2015).

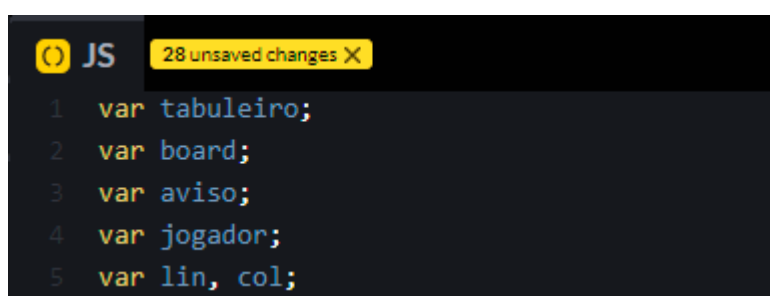
Fonte: <https://www.javascriptprogressivo.net/2019/01/Codigo-Completo-Jogo-velha-JavaScript-HTML-JS.html>

Como o código é relativamente grande, vamos separar o HTML do JS e começar analisando o HTML:



Link para ter acesso ao código: <https://github.com/GRANCodigo/PraticaDeProgramacao/blob/8943797a40f131f9365dcae9e917018e1d707212/Unidade3Aula5a>

No nosso HTML temos a tag <body> chamando a função inicial() através do evento onload. O evento onload não exige interação com o usuário, pois ele ocorre quando um objeto ou recurso tenha sido completamente carregado. Dentro do <body> temos uma div com id=board e é nela que vamos exibir o tabuleiro do jogo. Em seguida temos dois campos de formulário, onde o jogador vai indicar a linha e a coluna que deseja marcar. Como nosso tabuleiro é 3x3, os jogadores só poderão escolher números de 1 até 3. Depois de escolher o jogador clica em “Marcar” que por sua vez chama a função jogar(). A última tag do nosso body é uma div com id=aviso, onde será exibido a vez de cada jogador e também a mensagem que anunciará o jogador vencedor. Agora vamos ver o código em JavaScript que vai fazer o jogo acontecer. Vamos por partes:



O primeiro passo é declarar as variáveis que serão utilizadas fora do escopo de qualquer função, são elas:

- **tabuleiro:** responsável por armazenar a matriz 3x3, inicializada com todos os elementos iguais a 0. Regra: quando o primeiro jogador marca uma posição, mudamos de 0 para 1. Quando o segundo jogador marca uma posição, mudamos de 0 para -1;
- **board:** a variável board vai se conectar com a div board;
- **aviso:** a variável aviso vai se conectar com a div aviso;

- jogador: esta variável será incrementada a cada jogada. Para saber de quem é a vez de jogar, vamos utilizar o MOD(%) dela por 2, daí basta somar 1 ao resultado que teremos sempre os valores 1 e 2 se alternando;
- *lin*: responsável por pegar a posição da linha;
- *col*: responsável por pegar a posição da coluna.

A função `inicia()` roda apenas quando a página é carregada. Ela é responsável por criar um array de arrays (*linha 8*), ou seja, um array multidimensional. Nas *linhas 9 e 10*, as variáveis `board` e `aviso` são linkadas com o HTML. Na *linha 11*, o jogador é inicializado como 1. Em seguida, todos os elementos de array de arrays são inicializados com 0.

```

7 * function inicia(){
8   tabuleiro = new Array(3);
9   board = document.getElementById('board');
10  aviso = document.getElementById('aviso');
11  jogador = 1;
12
13  for(let i=0 ; i<3 ; i++)
14    tabuleiro[i] = new Array(3);
15
16  for(let i=0; i<3 ; i++)
17    for(let j=0 ; j<3 ; j++)
18      tabuleiro[i][j]=0;
19  exibe();
20 }

```

A função `exibe()` é responsável por criar a estrutura da tabela (linhas, colunas) e armazenar na variável HTML. A tabela aparece na nossa página graças ao `innerHTML` na *linha 37*.

Para saber mais sobre tabelas acesse: [https://www.w3schools.com/html/html\\_tables.asp](https://www.w3schools.com/html/html_tables.asp). As *linhas 27-33* exibem “—” caso encontre o valor 0 no tabuleiro, “X” caso o valor seja 1 e “O” caso o valor seja -1. Lembra da regra da variável `tabuleiro` descrita acima?

LINK



```

22 * function exibe(){
23   HTML = '<table cellpadding="10" border="1">';
24 *   for(let i=0; i<3 ; i++){
25     HTML += '<tr>';
26     for(let j=0 ; j<3 ; j++)
27       if(tabuleiro[i][j]==0)
28         HTML += '<td>  _  </td>';
29     else
30       if(tabuleiro[i][j]==1)
31         HTML += '<td> X </td>';
32     else
33       HTML += '<td> 0 </td>';
34     HTML += '</tr>';
35   }
36   HTML += '</table><br/>';
37   board.innerHTML = HTML
38 }

```

Primeiramente, a função jogar() exibe na tela o aviso de quem é a vez de jogar (*linha 42*). Em seguida, ela pega os valores de linha e coluna digitados pelo usuário. Lembre que tudo que é digitado num campo de formulário é considerado como string? Então, precisamos pegar essas string e converter em números (*linhas 43 e 44*). O usuário escolhe entre os números 1, 2 e 3, porém o tabuleiro contém os índices com posições 0, 1 e 2, por isso é necessário subtrair 1 do valor que foi escolhido ou digitado pelo usuário.

```

40 function jogar()
41 * {
42   aviso.innerHTML='Vez do jogador: ' + ((jogador)%2 + 1);
43   lin = parseInt(document.getElementById("lin").value)-1;
44   col = parseInt(document.getElementById("col").value)-1;
45
46   if(tabuleiro[lin][col]==0)
47     if(jogador % 2)
48       tabuleiro[lin][col] = 1;
49     else
50       tabuleiro[lin][col] = -1;
51 *   else{
52     aviso.innerHTML='Campo ja foi marcado!'
53     jogador--;
54   }
55   jogador++;
56   exibe();
57   checa();
58 }

```

Para marcar "X" ou "O" é necessário verificar se a posição está vazia ou não no tabuleiro[lin][col]==0, se estiver vazia e for o primeiro jogador coloca-se 1 ("X") na posição, se for o segundo coloca-se -1 ("O") na posição. Caso não esteja vazio (else), a mensagem "**campo já foi marcado**" é exibida na tela e subtraímos 1 da variável jogador (linha 52). Ao encerrar o bloco if-else a variável jogador é incrementada para que o próximo possa realizar a sua jogada. Além disso, a função jogar() também invoca as funções exibe() e checa().

A função checa() é responsável por checar quem ganhou o jogo. Para isso, ela percorre todas as linhas, colunas e diagonais do tabuleiro para verificar se a soma é igual a 3 ou -3.

```
60 function checa()
61 {
62     var soma;
63
64     //Linhas
65     for(let i=0 ; i<3 ; i++){
66         soma=0;
67         soma=tabuleiro[i][0]+tabuleiro[i][1]+tabuleiro[i][2];
68
69         if(soma==3 || soma== -3)
70             aviso.innerHTML="Jogador " + ((jogador)%2 + 1) + " ganhou! Linha " + (i+1) + " preenchida!";
71     }
72
73     //Colunas
74     for(let i=0 ; i<3 ; i++){
75         soma=0;
76         soma=tabuleiro[0][i]+tabuleiro[1][i]+tabuleiro[2][i];
```

```
78         if(soma==3 || soma== -3)
79             aviso.innerHTML="Jogador " + ((jogador)%2 + 1) + " ganhou! Coluna " + (i+1) + " preenchida!";
80     }
81
82     //Diagonal
83     soma=0;
84     soma = tabuleiro[0][0]+tabuleiro[1][1]+tabuleiro[2][2];
85     if(soma==3 || soma== -3)
86         aviso.innerHTML="Jogador " + ((jogador)%2 + 1) + " ganhou! Diagonal preenchida!";
87
88     //Diagonal
89     soma=0;
90     soma = tabuleiro[0][2]+tabuleiro[1][1]+tabuleiro[2][0];
91     if(soma==3 || soma== -3)
92         aviso.innerHTML="Jogador " + ((jogador)%2 + 1) + " ganhou! Diagonal preenchida!";
93 }
```



O primeiro for é responsável por verificar as colunas, o segundo por verificar as linhas e o terceiro por verificar as diagonais. Se o resultado da soma for igual a 3 ou -3 é exibida uma mensagem indicando qual a linha, coluna o diagonal aquele jogador venceu. Veja:

X	X	O
—	O	X
O	O	X

Linha :  Coluna:  Marcar  
Jogador 2 ganhou! Diagonal preenchida!

## 2. EXEMPLO 2 – RECURSIVIDADE

Para encerarmos esta aula, vamos fazer mais um exemplo de recursividade. Um outro exemplo clássico de recursividade é a sequência de Fibonacci. Ela é composta por uma sucessão de números que tem como primeiros termos os números 0 e 1 e, a seguir, cada termo subsequente é obtido pela soma dos dois termos predecessores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597...

Fórmula matemática Fibonacci:  $F_n = F_{n-1} + F_{n-2}$ .

Vamos criar um script para imprimir no console os 10 primeiros números da sequência de Fibonacci utilizando a recursividade.

```

1  function Fibonacci(n) {
2      if (n == 0)
3          return 0;
4      if (n == 1)
5          return 1;
6
7      return Fibonacci(n - 1) + Fibonacci(n - 2);
8  }
9
10 for (var n=0; n<10; n++){
11     console.log(Fibonacci(n));
12 }
13

```

Console

```

0
1
1
2
3
5
8
13
21
34

```

Repare que aqui foi necessário chamar a função Fibonacci dentro do console.log, que por sua vez está dentro de uma estrutura de repetição para que a sequência fosse impressa.

O conteúdo deste livro eletrônico é licenciado para GLEITON - 08303020692, vedada, por quaisquer meios e a qualquer título, a sua reprodução, cópia, divulgação ou distribuição, sujeitando-se aos infratores à responsabilização civil e criminal.

## CONSIDERAÇÕES FINAIS

Nesta aula tivemos a oportunidade de colocar a mão e aprender uma pouco mais sobre como implementar arrays de arrays, como definir e chamar funções e como trabalhar com recursividade. A ideia é que a partir de agora você consiga dar os seus próprios passos. Aproveite para acessar, explorar e editar o código do jogo da velha que está disponível na apostila. JavaScript não é difícil, mas a prática se faz necessário. Sendo assim, deixo a disposição de vocês um link ([https://oprogramador.bsb.br/aprenderjs\\_exercicios.php?page=1](https://oprogramador.bsb.br/aprenderjs_exercicios.php?page=1)) com exercícios resolvidos para que vocês possam testar ainda mais os conhecimentos adquiridos em nossas aulas. Bons estudos!

## MATERIAIS COMPLEMENTARES

Javascript – recursão, fatorial e fibonacci:

<https://youtu.be/H5u1dOoCPnc>

## REFERÊNCIAS

DE MATOS, F. J. M. JavaScript Progressivo: *Curso completo de JavaScript para iniciantes*, 2015. Disponível em: <<https://www.javascriptprogressivo.net/>>. Acesso em: 20 de nov. de 2022.