

# Funções

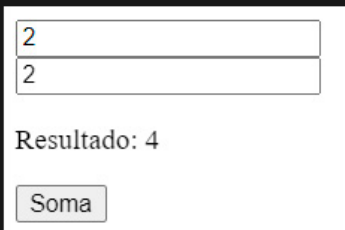
## Objetivo da Aula

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);
- **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

Diagrama da sintaxe de uma função de declaração sem parâmetros:

```
function name() {}
```

As partes são rotuladas como:

- palavra reservada:** `function`
- nome da função:** `name`
- parenteses:** `()`
- chaves:** `{}`

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

Diagrama da sintaxe de uma função de declaração com parâmetros:

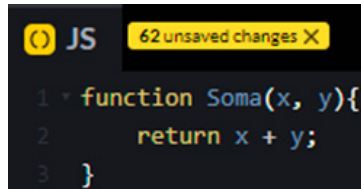
```
function name(parametro1, parametro2) {}
```

As partes são rotuladas como:

- palavra reservada:** `function`
- nome da função:** `name`
- parenteses:** `()`
- primeiro parâmetro:** `parametro1`
- segundo parâmetro:** `parametro2`
- chaves:** `{}`

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

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



```

1 * function Soma(x, y){
2     return x + y;
3 }

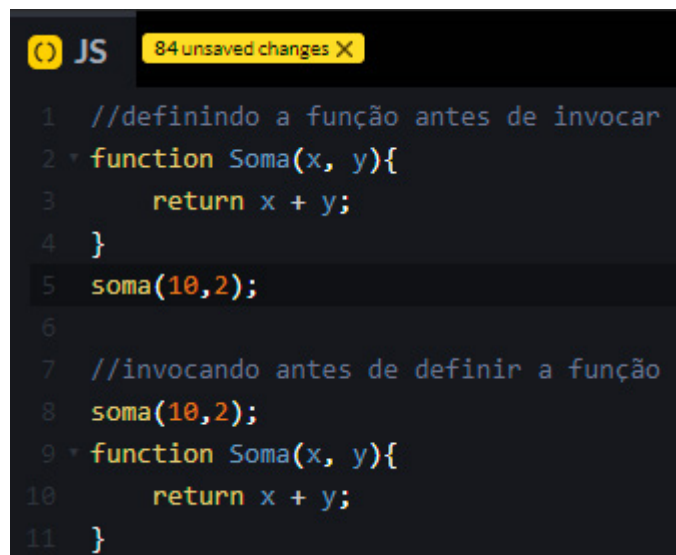
```

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:

```
JS 97 unsaved changes X
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

Diagrama explicando a sintaxe de uma função de expressão com parâmetros:

```
const name = function(parametro1, parametro2) {}
```

As partes são identificadas por setas:

- palavra reservada:** aponta para `const`.
- nome da variável:** aponta para `name`.
- palavra reservada:** aponta para `=`.
- parenteses:** aponta para os parênteses abertos e fechados da função.
- primeiro parâmetro:** aponta para `parametro1`.
- segundo parâmetro:** aponta para `parametro2`.
- chaves:** aponta para as chaves de abertura e fechamento da função.

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

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

```
JS 100+ unsaved changes X
1 * let somatorio = function (x,y){
2 *   return x + y;
3 * }
```

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:

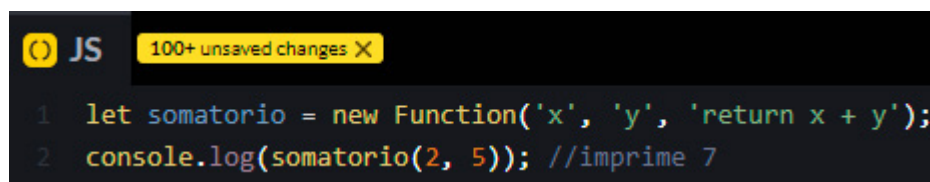
```

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 (*linha1*) 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:

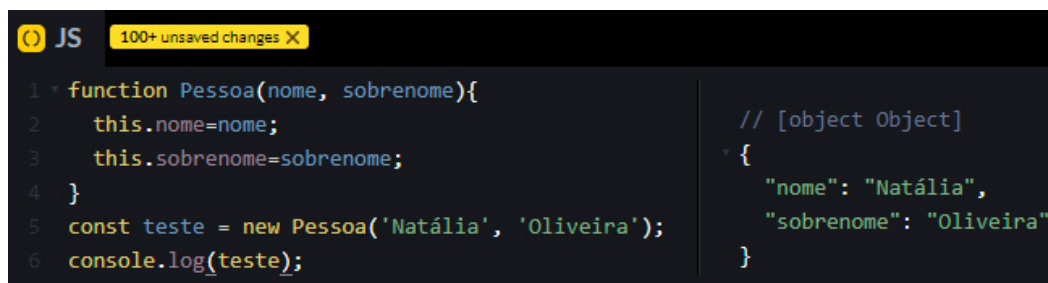


```

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):



```

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:

The screenshot shows a code editor with a JavaScript file named 'testandoGeradora.js'. The code defines a generator function `testandoGeradora()` that yields the values 1, 2, and 3. It then creates a variable `funcaoGeradora` pointing to the function and calls `next()` three times, logging the results to the console.

```

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 three objects returned by `next()`:

```

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

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

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

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

```

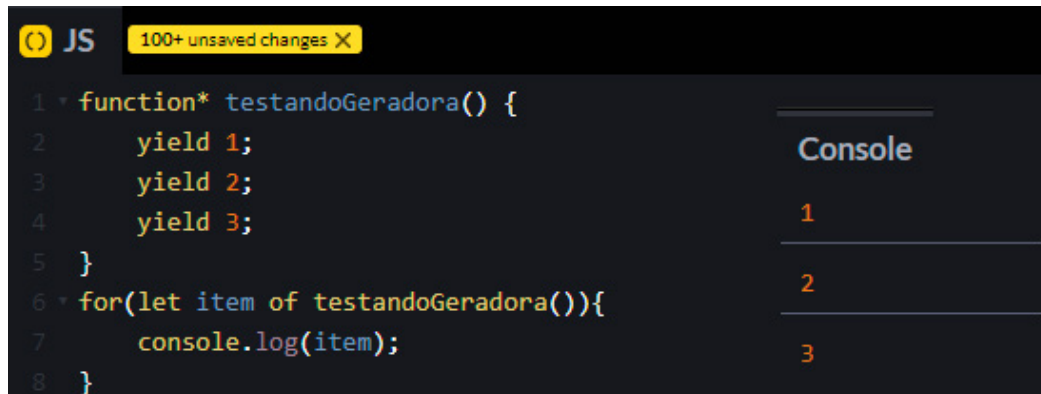
Vale ressaltar que uma função geradora quando invocada sempre retorna um **objeto iterador**. O que isso significa? Conforme a página oficial da MDN (2022),

(...) em JavaScript um **iterador** 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/dlaOlyCwbYk>



## 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.