

## VÍDEO 31 (aula 17) - Funções (básico)

Funções executam ações: imagina que temos um trecho, um bloco de código, que executa alguma ação.

A única coisa que queremos com a função é mandar um valor pra ela, e geralmente pegar um valor de volta, mas tem funções que não retorna valores também.

O nome das funções tem as mesmas regras das variáveis. Geralmente, o nome da função é a ação que ela vai executar.

Quando declaramos a função, não precisamos do ponto e vírgula.

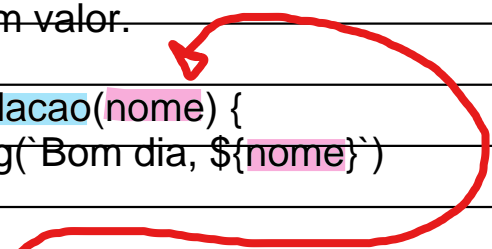
Fica assim:

```
function saudacao() {  
  console.log('Bom dia!')  
}
```

saudacao(); -> Chamando a função para que ela seja executada.

Suponha que eu queira dar 'Bom dia' e o nome da pessoa. Para isso, posso criar parâmetros, que vem dentro dos (). Dentro desse parâmetro, minha função pode receber algum valor.

```
function saudacao(nome) {  
  console.log('Bom dia, ${nome}')  
}
```



saudacao('Luiz'); -> chamada; esse argumento vai subir lá pro parâmetro da função

Para passarmos o valor para o parâmetro, colocamos o valor dentro da nossa chamada. Quando eu chamo a função, eu digo o que eu quero que esteja dentro do parâmetro.

Esse valor vai ser recebido dentro do nosso parâmetro e posso utilizar esse parâmetro em qualquer lugar no corpo da minha função.

Essa função é reutilizável e posso utilizar ela em qualquer lugar do meu código e podemos passar qualquer valor pra dentro do parâmetro:

```
function saudacao(nome) {  
  console.log('Bom dia, ${nome}!')  
}
```

```
saudacao('Luiz');  
saudacao('Maria');  
saudacao('Felipe');
```

Resultado:

Bom dia, Luiz!

Bom dia, Maria!

Bom dia, Felipe!

Também podemos fazer o seguinte:

```
const variavel = saudacao('João');  
console.log(variavel);
```

Resultado:

Bom dia, João!

undefined

Quero salvar o que a função retorna dentro da variavel.

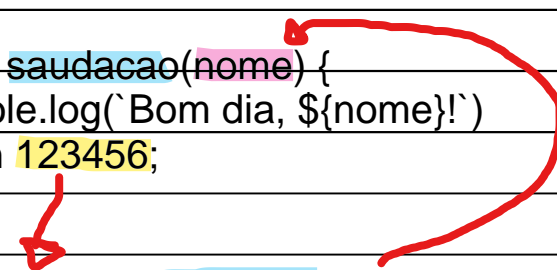
Só que essa função não está retornando nada, apenas está exibindo algo na tela.

Por padrão qualquer função criada no JS vai retornar undefined caso não especificarmos o que queremos que ela retorne.

Mesmo assim ela foi executada, pois chamamos ela com os (), exibiu o valor na tela, mas não retornou nenhum valor.

Se eu quero que minha função faça alguma coisa e também retorne um valor, utilizamos a palavra 'return'. Fazemos assim:

```
function saudacao(nome) {  
  console.log(`Bom dia, ${nome}!`)  
  return 123456;  
}
```



```
const variavel = saudacao('João');  
console.log(variavel);
```

Desse modo, estamos dizendo o que queremos que a função retorne. Ela vai continuar sendo executada, vai exibir na tela com o console.log e vai retornar um valor na tela. Esse return vai ser guardado dentro da nossa variavel.

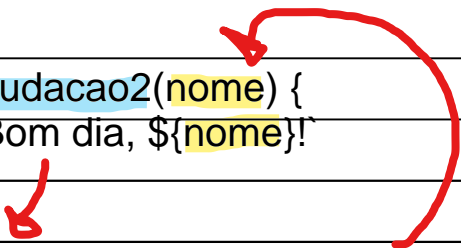
Resultado:

Bom dia, João!

123456

Podemos também apagar a linha do console.log e jogar a mensagem diretamente dentro do nosso return. Fica assim:

```
function saudacao2(nome) {  
  return `Bom dia, ${nome}`!  
}
```

A red arrow originates from the return statement in the function definition and points to the argument 'saudacao2('José')' in the function call below.

```
const variavel2 = saudacao2('José');  
console.log(variavel2);
```

Resultado:

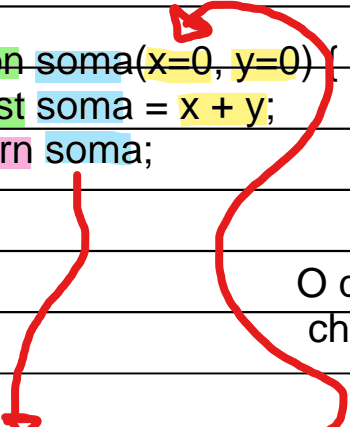
Bom dia, José!

Nossa variável recebeu o valor que foi retornado e depois mandamos exibir esse valor na tela.

#### ===== OUTRO EXEMPLO =====

Vamos criar uma função que faça a soma entre 2 valores. Fica assim:

```
function soma(x=0, y=0) {  
  const soma = x + y;  
  return soma;  
}
```

A red arrow originates from the return statement in the function definition and points to the argument 'soma(5, 15)' in the function call below.

Chamamos a função e passamos os parâmetros, que vão para dentro da nossa função.

O que for retornado na soma, vai para dentro da nossa chamada e vai ficar dentro da nossa variável.

```
let resultado = soma(5, 15);  
console.log(resultado); // Resultado: 20  
console.log(soma(10, 20)); // Resultado: 30
```

Obs. IMPORTANTE: Tudo que está dentro do bloco da função está protegido, não conseguimos acessar a const soma de fora, por exemplo.

Porém, o que está dentro da função não tem nada a ver com o que está fora.

Poderíamos, inclusive, declarar 2 variáveis const resultado (uma dentro e outra fora da função) que não teria problema.

Outra coisa que devemos notar é que, a partir do momento que o JavaScript encontrar a palavra 'return', nada abaixo dele será executado, ele vai encerrar a execução do programa por ali.

Na função acima, se mandarmos fazer a conta sem enviar nenhum parâmetro, vai retornar NaN.

Obs.: Geralmente, vamos querer criar funções pequenas que tem só uma função, que só tem 1 trabalho

VER MAIS NOS CÓDIGOS DA AULA 17, function1.js, function2.js e function3.js, PARA SABER MAIS SOBRE FUNÇÕES.

### VÍDEO 32 (aula 18) - Objetos (básico)

O objeto facilita na hora de agrupar coisas relacionadas em uma única variável, por exemplo. Para criar OBJETO = {} / Para criar ARRAY = []

Em vez de fazermos:

```
const nome01 = 'Luiz';  
const sobrenome01 = 'Miranda';  
const idade01 = 25;
```

Podemos fazer:

```
const pessoa1 = {  
  nome: 'Luiz',  
  sobrenome: 'Miranda',  
  idade: 25  
};
```

Para acessarmos determinado atributo, utilizamos a notação de ponto(.):  
console.log(pessoa1.nome) // Resultado: Luiz

VER MAIS NOS CÓDIGOS DA AULA 18, objeto1.js e objeto2.js PARA SABER MAIS SOBRE OBJETOS.

## VÍDEO 33 (aula 19) - Valores primitivos e valores por referência

Tipos de dados PRIMITIVOS (valores imutáveis): string, number, boolean, undefined, null, bigint, symbol - Não podemos mudar eles.

Exemplo:

```
let nome = 'Luiz';  
nome = 'Otávio';  
console.log(nome) // Resultado: Otávio
```

Neste caso, não estamos mudando o dado primitivo, estamos mudando o valor da variável

**VER CÓDIGOS DA AULA 19 PARA SABER MAIS**

## VÍDEO 34 (aula 20) - Exercício para função, array e objetos

Vamos ter um formulário onde a pessoa vai digitar o nome, sobrenome, peso e altura.

Só vamos exibir os valores que forem digitados no formulário e também vamos salvar tudo o que for digitado dentro de um array que contém vários objetos.

Uma coisa que devemos nos atentar é que tudo que acontece dentro do navegador é considerado um evento. Por exemplo: evento de clique do mouse, evento de movimento do mouse, etc. Tem evento pra praticamente tudo no navegador.

Nosso trabalho como desenvolvedor é colocar algum espião nesse evento dentro do nosso código. Por exemplo: posso falar pro JS: quando o mouse se mexer, faça isso, quando a pessoa digitar alguma coisa, faça isso, quando o formulário for enviado (evento de submit) posso capturar esse evento e falar que não quero que ele atualize a página, etc..

Existe uma maneira da gente proteger nosso código para que ele não polua o escopo global e nem sofra influências externas: é só envolver tudo dentro de uma função. Veja:

```
function meuEscopo() {  
  let nome = 'Luiz';  
  alert(1);  
}
```

Assim, nada que está dentro da função poderá ser acessado no escopo global.

\_\_\_/\_\_\_/\_\_\_

Note que ao acessarmos a página, nossa função não vai ser executada. Para ela executar, temos que chamar a função, botar ela pra executar. Fazemos assim:

meuEscopo();

Agora a única coisa do meu código que está poluindo o escopo global é nossa função meuEscopo(). ~~Tudo que estiver dentro da nossa função estará protegido.~~

Obs.: temos um jeito de criar uma função que ela é auto-invocada. Ela é criada e auto-invocada ao mesmo tempo, mas veremos isso mais pra frente no nosso curso.

Sabendo disso, vamos começar a trabalhar dentro da função, não vamos mais trabalhar no escopo global.













