

# Funções em JavaScript



# Funções

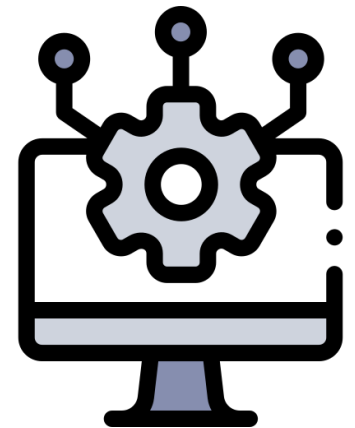
---

De um modo geral, uma função é um "**subprograma**" que pode ser chamado por código externo (ou interno no caso de recursão) à função.

- ❑ Funções são uma parte fundamental da linguagem JavaScript e são usadas para encapsular um conjunto de instruções em um único bloco reutilizável.

Assim como o próprio programa, uma função é composta por uma sequência de instruções chamada de **corpo da função**.

- ❑ Valores podem ser passados para uma função, e ela pode retornar um valor.

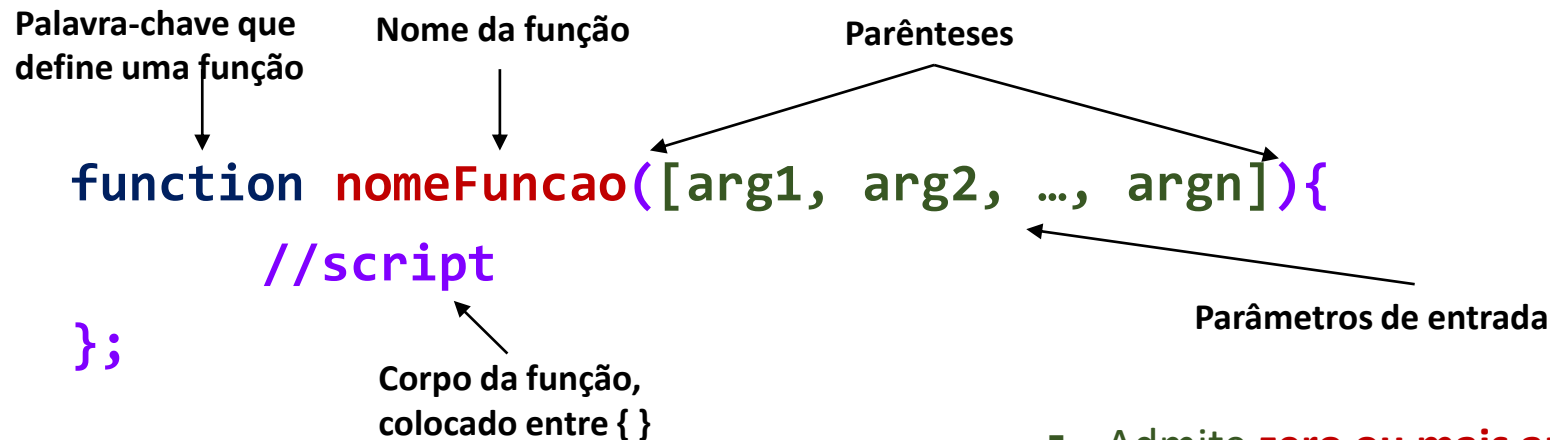




# function: Objetivos e Sintaxe

Funções visam **encapsular blocos de códigos** que possuem objetivos bem definidos.

Sintaxe  
geral



Exemplo

```
function multiplica(x, y){  
    const resultado = x * y;  
    alert(resultado);  
};
```

- Admite **zero ou mais argumentos**;
- Define um **bloco de código** a ser executado em determinados pontos do script;
- Útil para **não reescrever códigos repetidamente**;
- Permite que a função seja “chamada” **sempre que necessário**.



# function: Nomes e içamento (*hoisting*)

A criação de uma function, **segue as mesmas regras de nomes, utilizadas na definição de identificadores de variáveis.**

Funções declaradas por meio da palavra-chave **function**, sofrem o ***hoisting*** (elevação ou içamento).

O ***hoisting*** permite a utilização da função, mesmo antes da aparição de seu corpo no código.

```
console.log(soma(3, 8));
```

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



Recordando... A *engine* do **JavaScript** iça as definições de **function** e variáveis declaradas com **var**, para o topo do arquivo no momento da execução.



# Retornos de uma function

---

```
function multiplica(x, y){  
    let resultado = x * y;  
    console.log(resultado);  
};  
multiplica(10, 5);
```

## Funções void

- ❑ Executam instruções dentro delas;
- ❑ Não retornam nada para o ponto do código que chamou a função;
- ❑ Não é possível utilizar os valores posteriormente.

```
function multiplica(x, y){  
    let resultado = x * y;  
    return resultado;  
};  
let r = multiplica(10, 5);  
console.log("O resultado é " + r);
```

## Funções com return

- ❑ Permitem retornar valores para o ponto do código que chamou a função.

# *First class objects*

---





# First class objects

Em **JavaScript**, funções são **objetos de primeira classe**.

Esta característica **permite tratar as funções como dados**, ou seja, elas podem ser passadas como **parâmetro para outra função**, ou ser o **retorno de uma função**.

```
const souUmDado = function(){  
    console.log('Sou um dado.');
```

```
};  
souUmDado();
```

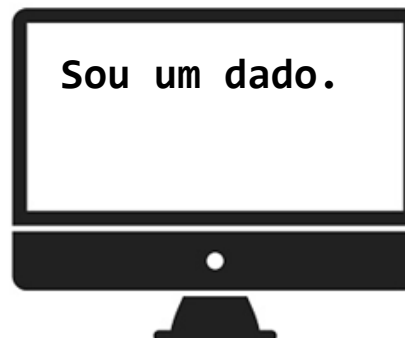
## function expressions

Funções nas quais jogamos o resultado de sua execução para uma variável.

A variável recebe uma função... A partir daí **ela passa a ser uma função**.

Uma **function expression** é muito similar e tem quase a mesma sintaxe de uma declaração de **function regular**

A principal diferença entre ela e uma função regular, é o nome da função, o qual pode ser omitido para criação de **funções anônimas**.



Exemplo 02



# First class objects

Em **JavaScript**, funções são **objetos de primeira classe**.

Esta característica **permite tratar as funções como dados**, ou seja, elas podem ser passadas como **parâmetro para outra função**, ou ser o **retorno de uma função**.

```
const souUmDado = function(){  
    console.log('Sou um dado.');
```

};

```
souUmDado();
```

Possibilitar o armazenamento do resultado de uma expressão em uma variável é um recurso poderoso, pois **agora é possível passar esta variável como um parâmetro de uma outra função**.

```
function executaFuncao(funcao){  
    funcao();  
}
```

Recebendo uma função como parâmetro de entrada

```
executaFuncao(souUmDado);
```

Executando a função que foi passada como argumento.

Passando uma função **como parâmetro de outra**.







# *First class objects*

---

Em **JavaScript**, funções são **objetos de primeira classe**.

Característica de linguagens de programação que tratam as funções da mesma forma que tratam outros tipos de dados, como números, strings e objetos.

**Isso significa que as funções em linguagens com suporte a funções de primeira classe podem ser:**

**Atribuídas a variáveis:** é possível atribuir uma função a uma variável, tornando-a referenciável por esse nome.

**Passadas como argumentos:** permite passar uma função como um argumento para outra função.

**Retornadas por outras funções:** Funções podem ser retornadas como valores de outras funções.

**Armazenadas em estruturas de dados:** possibilita armazenar funções em arrays, objetos e outras estruturas de dados.

**Serem criadas dinamicamente:** é possível criar funções dinamicamente em tempo de execução.



# 1) Armazenamento em variáveis

---

Funções podem ser armazenadas em variáveis e atribuídas a elas.

Isso significa que possível criar uma função e depois referenciar essa função por meio de uma variável.

```
const saudacao = function() {  
    console.log("Olá TSI!");  
};
```

```
saudacao(); // Chama a função armazenada na variável saudacao
```





## 2) Passagem como argumentos

---

**Funções podem ser passadas como argumentos para outras funções.**

Isso permite a criação de funções de ordem superior, onde uma função pode aceitar outras funções como parâmetros.

```
function executar(funcao) {  
    funcao();  
}
```

```
executar(function() {  
    console.log("Função passada como argumento.");  
});
```





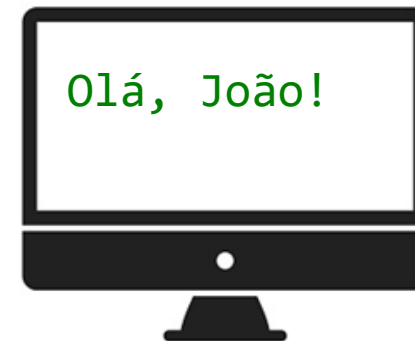
### 3) Retorno de outras funções

---

Funções podem ser retornadas de outras funções, permitindo a criação de funções que geram ou retornam outras funções.

```
function criarSaudacao(nome) {  
    return function() {  
        console.log(`Olá, ${nome}!`);  
    };  
}
```

```
const saudacaoJoao = criarSaudacao("João");  
saudacaoJoao(); // Olá, João!
```





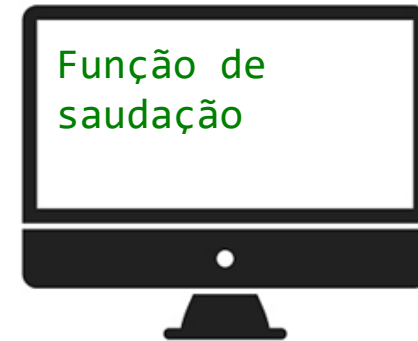
## 4) Propriedades e métodos

---

Funções podem ter propriedades e métodos, assim como qualquer outro objeto em JavaScript.

```
function saudacao() {  
    console.log("Olá!");  
}
```

```
saudacao.info = "Função de saudação"; // Adicionando uma propriedade  
console.log(saudacao.info); // Função de saudação
```





## 5) Criação dinâmica

---

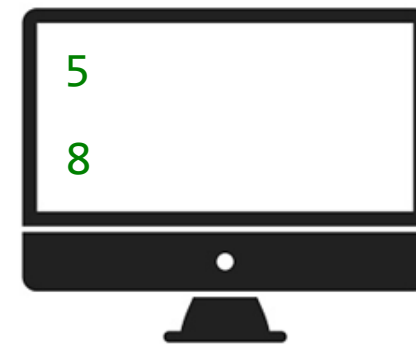
Funções podem ser criadas dinamicamente em tempo de execução usando o construtor `Function` ou com funções anônimas.

```
const soma = new Function('a', 'b', 'return a + b;');
```

```
console.log(soma(2, 3)); // 5
```

```
const soma = function(a, b) {  
    return a + b;  
};
```

```
// Usa a função criada  
console.log(soma(5, 3)); // 8
```





# Resumindo...

---

A característica de ser um "*first-class object*" significa que as funções em JavaScript podem ser manipuladas como qualquer outro objeto:

- armazenadas em variáveis;
- passadas como argumentos;
- retornadas de outras funções; e
- ter propriedades.

Isso torna as funções extremamente flexíveis e poderosas para criar abstrações e manipular o fluxo de controle em JavaScript.



# arrow functions => (funções de seta)

## Recurso introduzido no EcmaScript15 (ES6)

- Permite escrever versões “mais curtas” de **function expressions**.

```
// função declarativa
function potencia(n){
    return n ** 0.5;
};
```

```
// Função anônima
// function expression
const potencia = function(n){
    return n ** 0.5;
};
```

```
// arrow function
const potencia = (n) => n ** 0.5;
```

- 1) Não usa a palavra **function**
- 2) Não usa chaves
- 3) Não usa **return**

[Exemplo 03](#)

## Sintaxe

```
(param1, param2, ..., paramN) => { statements }
(param1, param2, ..., paramN) => expression
// equivalente a: => { return expression; }
```

// Parênteses são opcionais quando só há um parâmetro:

```
(singleParam) => { statements }
singleParam => { statements }
```

// Uma função sem parâmetros deve ser escrita com parênteses.

```
() => { statements }
```





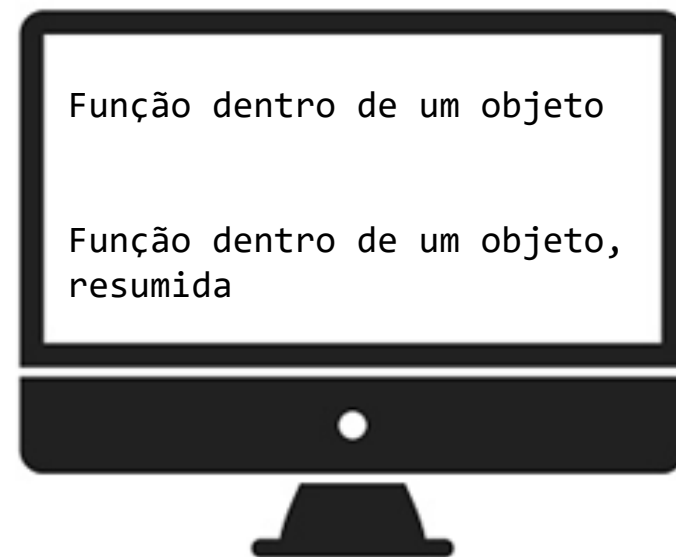
# Funções dentro de objetos

Em **JavaScript**, é possível criar funções dentro de **objetos**.

```
const obj = {  
  run: function(){  
    console.log('Função dentro de um objeto');  
  }  
}  
obj.run();
```

Nas versões mais recentes do JS é possível criar o mesmo método, sem a palavra **function** e sem o “:”

```
const obj2 = {  
  run(){  
    console.log('Função dentro de um objeto, resumida');  
  }  
}  
obj2.run();
```



# Parâmetros de funções

---





# Flexibilidade de parâmetros

---

**JavaScript** é mais flexível que outras linguagens de programação como Java ou PHP, por exemplo.

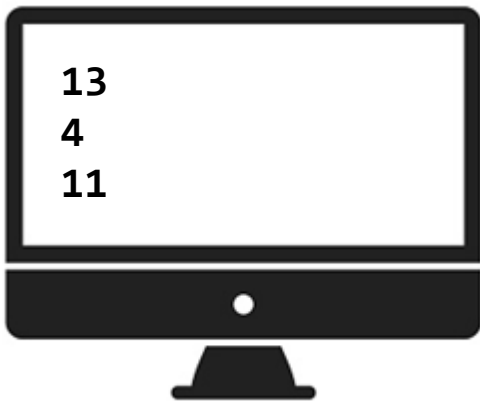
Mesmo estabelecendo quais os parâmetros que uma função recebe (assinatura), **não necessariamente esses argumentos precisam ser passados.**

Maior flexibilidade em termos de quantidade de parâmetros que podemos passar para a função, **sem resultar em erros.**



# Parâmetros de entrada padrão

É possível definir **valores padrão**, caso algum parâmetro não seja enviado.



Valores padrão para cada variável.

```
function soma(x = 10, y = 3) {  
    const resultado = x + y;  
    return resultado;  
}
```

```
console.log(soma());  
console.log(soma(1));  
console.log(soma(3, 8));
```

Exemplo



# Valores padrão

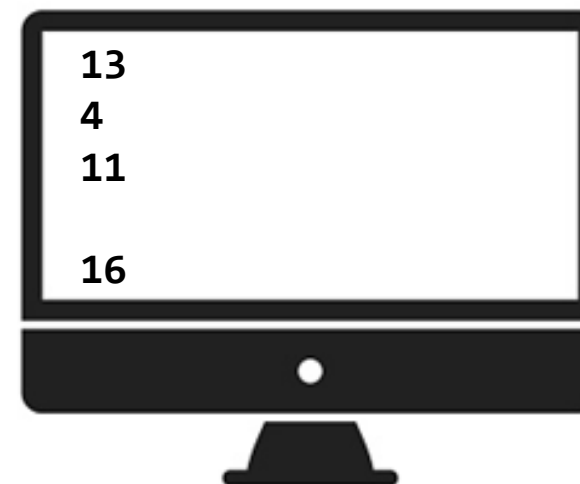
```
function soma( x = 10, y = 3 ) {  
    const resultado = x + y;  
    return resultado;  
}  
  
console.log(soma());  
console.log(soma(1));  
console.log(soma(3, 8));
```

E se for preciso que o primeiro valor assuma o valor padrão, e os demais fossem passados como argumentos?

Se define este valor como **undefined**

```
console.log(soma(undefined, 6));
```

É possível definir valores padrão, caso algum dos parâmetros não seja enviado.



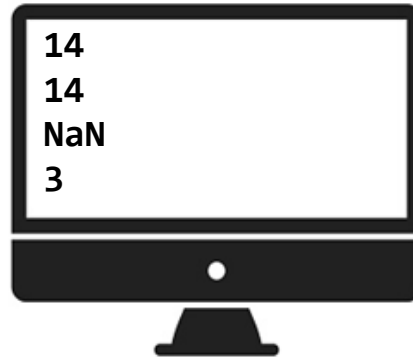
Em JavaScript setar valores padrão pode ser interessante, já que a linguagem **não obriga a determinação dos parâmetros formais nas funções.**

[Exemplo 06](#)



# Exemplo: flexibilidade de parâmetros

```
function soma(a, b){  
    let resultado = a + b;  
    return resultado;  
};
```



```
function soma(a, b){  
    if(b===undefined){  
        b = 0;  
    }  
    return a + b;  
};
```

Como tratar isso?

```
console.log(soma(7, 7));
```

```
console.log(soma(7, 7, 9, 18)); // JS desconsidera os parâmetros adicionais
```

```
console.log(soma(7));  
// Neste caso, JS acaba somando o primeiro valor com undefined  
// 7 + undefined = NaN (not a number)
```

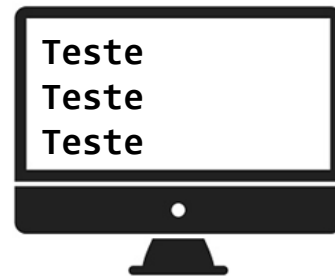


# arguments

Vamos dar uma olhada para entender para onde vão esses **argumentos extras não utilizados**.

```
function funcao(){
  console.log('Teste');
}
```

```
funcao();
funcao(1);
funcao('Valor', 2, 3, 4, 5, 6);
```



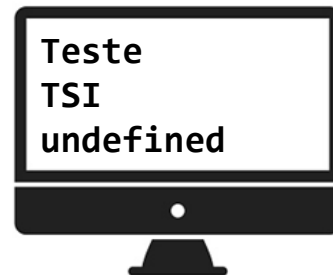
## Para onde foram esses argumentos?

Ao definir uma função com a palavra-chave **function**, temos a disposição o **arguments**, um **objeto que armazena todos os valores enviados para a função**.

É possível acessar esses dados por meio do seu índice.

```
function funcao(){
  console.log('Teste');
  console.log(arguments[1]);
  console.log(arguments[5]);
}
```

```
funcao(1, 'TSI', 5);
      0   1   2
```



**arguments** não funciona com **arrow functions**, apenas com **functions** ou **functions expressions**



# arguments

Vamos dar uma olhada para entender para onde vão esses argumentos extras não utilizados.

```
function funcao() {  
    let somatorio = 0;  
  
    for(let argumento of arguments){  
        somatorio += argumento;  
    }  
    console.log(somatorio);  
}  
  
funcao(1, 2, 3, 4, 5, 6);
```



Essa *feature* permite, **mesmo sem definir a espera de parâmetros formais**, passar e utilizar argumentos em uma função JavaScript.



Funções definidas com a palavra **function**, mantém uma variável especial **arguments**, que armazena todos os argumentos enviados.

[Exemplo 06](#)





# Trabalhando com os argumentos

Imagine que tenhamos uma situação em que precisamos desenvolver uma função como essa.

Os dois primeiros parâmetros definem a operação e o acumulador, e depois um conjunto de elementos sobre o qual se deseja operar.

```
function calculo(operador, acumulador, numeros){  
    console.log(operador, acumulador, numeros);  
}  
calculo('+', 0, [2, 3, 4, 5]);
```

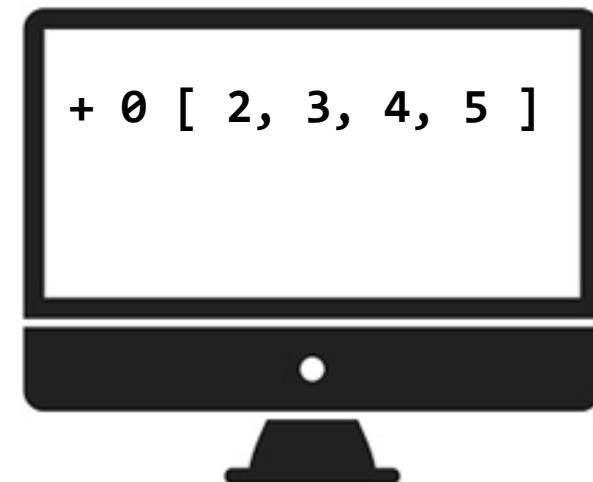
Operador

Acumulador

Números a operar

Ao invés de utilizar elementos na forma de vetor como terceiro elemento, é possível utilizar o **rest operator**.

...argumento



[Exemplo 07](#)

...rest operator





# rest operator ...

```
function calculo(operador, acumulador, numeros){  
    console.log(operador, acumulador, numeros);  
}
```

```
calculo('+', 0, [2, 3, 4, 5]);
```

Operador

Acumulador

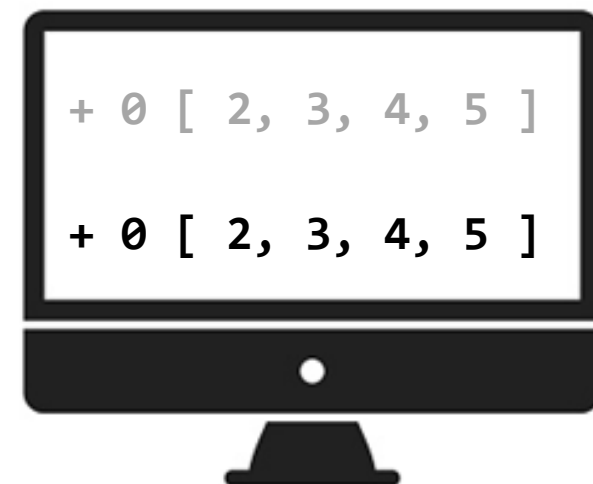
Números a operar

// rest operator

```
function cRest(operador, acumulador, ...números){  
    console.log(operador, acumulador, numeros);  
}  
  
cRest('+', 0, 2, 3, 4, 5);
```

Ao invés de utilizar elementos na forma de vetor como terceiro elemento, é possível utilizar o **rest operator**.

...argumento



[Exemplo 07](#)

O **rest operator** tem q ser o **último parâmetro formal da função**.



# Exemplo rest operator ...

```
function calculoRest(operador, acumulador, ...numeros)
{
  for(let numero of numeros){
    if(operador === '+') acumulador += numero;
    if(operador === '-') acumulador -= numero;
    if(operador === '*') acumulador *= numero;
    if(operador === '/') acumulador /= numero;
  }
  console.log(acumulador);
}
```

```
calculoRest('+', 0, 2, 3, 4, 5);
calculoRest('*', 1, 2, 3, 4, 5);
```

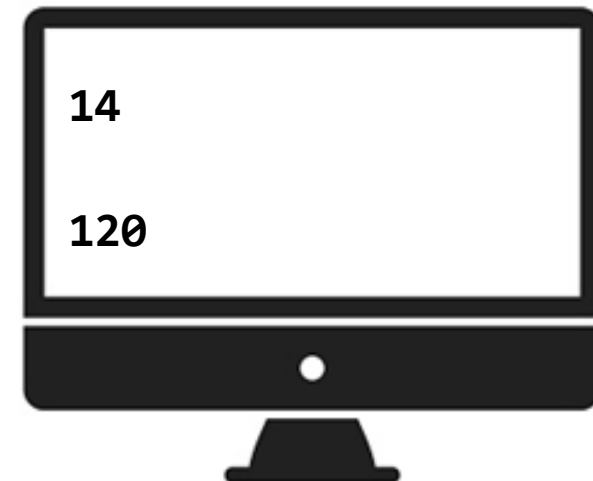
Operador

Acumulador

Números a operar

Ao invés de utilizar elementos na forma de vetor como terceiro elemento, é possível utilizar o **rest operator**.

...argumento



[Exemplo 07](#)

O **rest operator** tem q ser o **último parâmetro formal da função**.



# E com as arrow function?

Sempre que for necessário enviar uma quantidade de parâmetros indeterminados para uma função, é possível utilizar o **rest operator**.

```
const calculaArrow = (op, acum, ...nrs) => {  
  console.log(op, acum, nrs)  
}  
calculaArrow('+', 1, 6, 7, 8);
```

Terá o mesmo efeito de utilizar  
**function** ou **expression function**.



O **arguments** só existe em funções criadas com **function**.

```
const calculaArrow2 = (...args) => {  
  console.log(args);  
}  
calculaArrow2('+', 1, 6, 7, 8);  
                0   1   2   3   4
```

No entanto, o uso do **rest operator**,  
permite que tenhamos o mesmo  
**funcionamento do arguments**.

[Exemplo 08](#)

# Funções que retornam funções

---



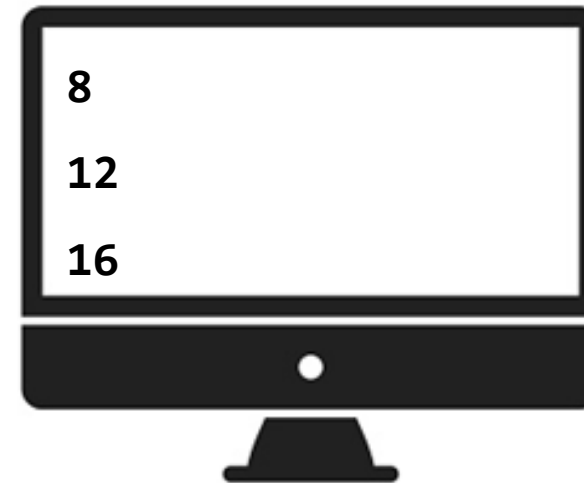


# Funções que retornam valores

Funções utilizam o **return** para retornar valores

```
function duplica(valor){  
    return valor * 2;  
}  
  
function triplica(valor){  
    return valor * 3;  
}  
  
function quadriplica(valor){  
    return valor * 4;  
}
```

```
console.log(duplica(4));  
console.log(triplica(4));  
console.log(quadriplica(4));
```



Exemplo

**return** pode retornar  
também outra função.



```
1
function criaMultiplicador(multiplicador){
  function multiplica(n){ 2
    return multiplicador * n;
  }
  return multiplica;
}
```

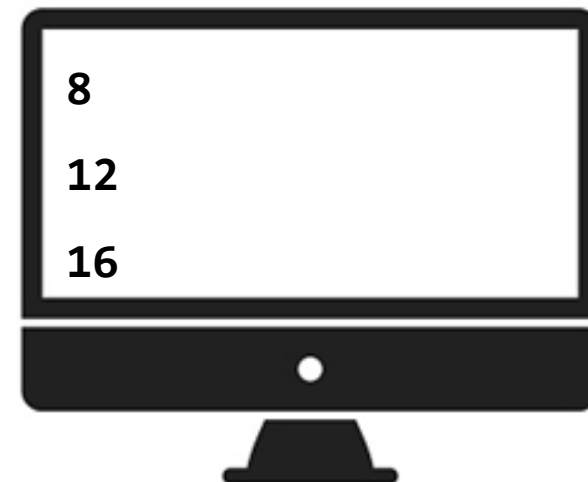
## O que foi desenvolvido?

- 1) Criamos uma **função** que recebe um parâmetro de entrada (criaMultiplicador);
- 2) Esse parâmetro é utilizado em uma **função interna** (multiplica);
- 3) Esta **função interna multiplica** é retornada no ponto do código em que foi chamada.
- 4) Podemos usar essa função como um *first class object*  
Característica que **permite tratar as funções como dados**, ou seja, elas podem ser passadas como **parâmetro**, ou ser o **retorno de uma função**.

```
3
const duplicador = criaMultiplicador(2);
const triplicador = criaMultiplicador(3);
const quadriplicador = criaMultiplicador(4);
```

```
console.log(duplicador(4));
console.log(triplicador(4));
console.log(quadriplicador(4));
```

4



Exemplo 08

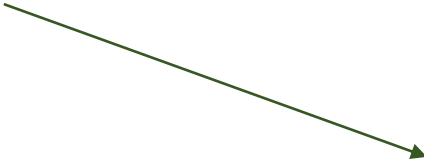




É possível **modificar** essa função:

- 1) Passar o **return para o início da função interna**;
- 2) Eliminar o nome da função interna, que **passa a ser anônima**.

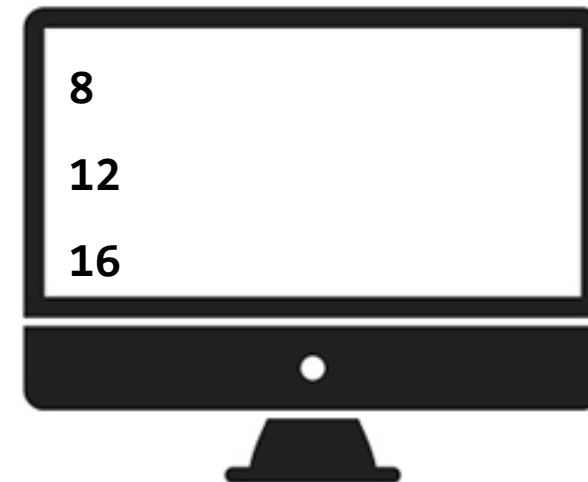
```
function criaMultiplicador(multiplicador){  
  function multiplica(n){  
    return multiplicador * n;  
  }  
  return multiplica;  
}
```



```
function criaMultiplicador(multiplicador){  
  return function(n){  
    return multiplicador * n;  
  };  
}
```

```
const duplicador = criaMultiplicador(2);  
const triplicador = criaMultiplicador(3);  
const quadriplicador = criaMultiplicador(3);
```

```
console.log(duplicador(4));  
console.log(triplicador(4));  
console.log(quadriplicador(4));
```



# Funções em JavaScript