

```
//FUNCTIONS:
//Podemos interpretar funções em Javascript como métodos(referência a POO), funções e módulos para o nosso sistema. No Javascript as
funções tem uma relevância muito grande, por isso é essencial que você entenda bem como elas funcionam no Javascript.

//FUNÇÃO QUE RECEBE PARÂMETROS PORÉM NÃO DÁ RETORNO:
function imprimirSoma(a, b) { //Por ser fracamente tipada não precisamos de definir tipos primitivos para os parâmetros, outro detalh
e é que no corpo de uma função são obrigatórios o uso das chaves (com excessão de arrow function)...
    console.log(a + b);
    //Mesmo quando uma função não tem o return declarado explicitamente nela, conceitualmente, ela possui o return sim, porém o seu r
eturn está com o valor "undefined", caso chamássemos pelo retorno dela literalmente iríamos ver o resultado como undefined...
}
console.log('Função recebe parâmetros mas não dá retorno...')
imprimirSoma(2, 3); //Perceba que passamos 2 argumentos para a função...
console.log(imprimirSoma(2, 3)) //Porém perceba que junto ao valor, toda função tem retorno, se não colocamos o resultado no retorno,
o retorno dela vém como undefined...


//O QUE ACONTECE QUANDO UMA FUNÇÃO NÃO RECEBE TODOS OS PARÂMETROS QUE DEVERIA:
console.log('\nFunção não recebeu todos os parâmetros que deveria...')
imprimirSoma(7); //Se a função deveria receber 2 argumentos, mas recebe só 1 ou outro será definido como undefined, podendo gerar um
resultado diferente do esperado


//O QUE ACONTECE QUANDO UMA FUNÇÃO RECEBE MAIS PARÂMETROS DO QUE DEVERIA:
console.log('\nFunção recebeu mais parâmetros do que deveria...')
imprimirSoma(3, 7, 2, 4, 4); //Perceba que a função vai usar somente os primeiros argumentos que correspondem ao seu número de parâme
tros, os demais serão ignorados...


//O QUE ACONTECE SE VOCÊ NÃO PASSA ARGUMENTO NENHUM:
console.log('\nFunção não recebe argumento nenhum...')
imprimirSoma(); //Perceba que ela não vai dar erro, simplesmente, por não ter valores para os seus parâmetros ela vai tratar todos os
argumentos como "undefined" e vai trazer um resultado específico de acordo com o tipo de dados da função...
```

//ALGUMAS FUNÇÕES O RESULTADO VAI DEPENDER DO VALOR PASSADO:

```
console.log('\nFunção recebe string no lugar de números e concatena eles...')
```

```
imprimirSoma('Gabriel ', 'Ferreira'); //Perceba que passamos strings que também utilizam o operador "+", por isso, nesse caso a função trouxe para nós um valor satisfatório, isso não acontece em todos os casos, vai depender da operação da função e do tipo de dados e em que passamos o valor da função.
```

//CRIANDO UMA FUNÇÃO QUE NÃO RECEBE PARÂMETROS NENHUM, MAS PODEMOS RECUPERAR PARÂMETROS PASSADOS NA CHAMADA COM ARGUMENTS:

//Toda função possui por padrão um array interno da linguagem chamado "arguments", nesse array ela guarda todos os parâmetros passados para ela, isso faz com que, mesmo que não tenhamos declarado nenhum parâmetro formal na assinatura da função ainda assim possamos recuperar parâmetros passados numa chamada da função usando o arguments...

```
function semParam(){ //Perceba que não recebemos parâmetro nenhum...
```

```
    let soma = 0; //essa variável irá somar ou concatenar qualquer valor passado como parâmetro...
```

```
    for (let i in arguments) { //o arguments vai guardar qualquer valor passado...
```

```
        soma += arguments[i]; //Aqui recuperamos os valores passados e os somamos a variável soma, ou concatenamos se forem strings..
```

```
    }
```

```
    return soma;
```

```
}
```

```
console.log("\nQuando a função não recebe parâmetro e mesmo assim passamos parâmetros na chamada da função e o recuperamos com arguments...")
```

```
console.log(semParam());
```

```
console.log(semParam(1));
```

```
console.log(semParam(1.1, 2.2, 3.3));
```

```
console.log(semParam(1.1, 2.2, "Teste"));
```

```
console.log(semParam('a', 'b', 'c'));
```

//FUNÇÃO COM RETORNO:

```
function soma(a, b=0) { //Perceba que nesse dessa vez demos um valor padrão para a função diretamente no campo de parâmetros
```

```
    return a + b; //Para a função ter retorno usamos a palavra reservada "return"...
```

```
}
```

```
console.log('\nFunção que utiliza retorno...')
```

```
console.log(soma(8, 2));
```

```
console.log(soma(7)); //Se não passamos o segundo valor como parâmetro ele pega o valor padrão da função...
console.log(soma()); //Se não passamos parâmetro nenhum para a função e todos os parâmetros dela não tiverem valor padrão ela retorna
rá um dos valores como "undefined" e nos devolverá um resultado estranho...
```

//DETALHE DO RETURN COM OBJECT:

//Essa é uma função que determina que: Se recebermos 2 salários vamos comprar uma TV de 50" e tomar sorvete, mas se recebermos 1 salário só vamos comprar 1 TV de 32" e tomar sorvete, porém, se não recebermos nada não vamos comprar TV nenhuma nem comprar sorvete...

```
function compras(salario1, salario2) {
  const comprarSorvete = salario1 || salario2; //Independente do salário, se recebermos pelo menos 1 vamos tomar sorvete...
  const comprarTV50 = salario1 && salario2; //A TV de 50" só será comprada se recebermos 2 salários...
  const comprarTV32 = !(salario1 ^ salario2); //A TV de 32 será comprada só se recebermos 1 salário ou outro...
  const manterSaudavel = !comprarSorvete; //Vamos manter nosso corpo saudável só se não comprarmos sorvete nenhum...
  return { //Perceba que no retorno de funções, quando criamos objetos, á partir de variáveis dentro da função, não precisamos de r
ferenciar seus nomes para depois capturar os seus valores, por padrão podemos colocar somente o nome da variável que automaticamente
ela se transforma na nossa chave e o seu valor vêm acompanhado dela...
    comprarSorvete,
    comprarTV50,
    comprarTV32,
    manterSaudavel}
}
```

```
console.log('\nQuando funções retornam objetos...')
console.log('\n6) Salario1 = true e Salario2 = true', compras(true, true));
console.log('\n7) Salario1 = true e Salario2 = false', compras(true, false));
console.log('\n8) Salario1 = false e Salario2 = true', compras(false, true));
console.log('\n9) Salario1 = false e Salario2 = false', compras(false, false));
```

//PARÂMETRO COM VALOR PADRÃO:

```
function tamanho(a=5){ //Criamos uma função que retorna um valor padrão de 5 independente se passamos um valor como parâmetro ou não.
```

```
..
  return console.log(a);
}
```

```
console.log('\nFunção que já possui valor padrão...')
```

```
tamanho();//Veja que mesmo sem passar valor nenhum, ela por padrão já tem um valor de 5
```

```
tamanho(10); //Quando passamos parâmetros ela respeita o valor passado no argumento
```

```
//GERANDO UM VALOR PADRÃO EM UMA VARIÁVEL QUE FOI DECLARADA DENTRO DE UMA FUNÇÃO:
```

```
function fazerVezes3(a) { //Se nenhum valor for passado ela somará 1 vezes 3...
```

```
    let n1 = a || 1; //Podemos escolher um valor padrão para uma variável que foi declarada dentro de uma função usando o operador ||  
    , onde, se a primeira opção for "false" o operador vai considerar a segunda opção...
```

```
    let n2 = a || 1; //Perceba que esse método só é bom quando aplicado a uma variável que foi declarada dentro de uma função, quando  
    desejamos atribuir um valor padrão diretamente a um argumento passado como parâmetro é mais fácil usar a declaração de valor padrão  
    diretamente no campo de parâmetros da função...
```

```
    let n3 = a || 1;  
    return console.log('A soma dos valores passados é:', n1 + n2 + n3);  
}
```

```
console.log('\nQuando desejamos declarar um valor padrão sobre uma variável declarada dentro de uma função...')
```

```
fazerVezes3(); //Perceba que quando nenhum valor é passado os valores das variáveis internos seguem o padrão...
```

```
fazerVezes3(2); //Mas, quando o valor é passado ela usa o valor passado como parâmetro...
```

```
//PASSANDO UMA FUNÇÃO COMO PARÂMETRO PARA UMA VARIÁVEL:
```

```
const somando = function (a, b) { //o Javascript possibilita o armazenamento de uma função dentro de uma variável ou constante, para  
isso note que a função não recebe nome, já que ela será referenciada pelo nome da variável
```

```
    console.log(a + b);  
}
```

```
console.log('\nFunção é passada como parâmetro de uma variável...')
```

```
somando(10, 10); //Perceba que usamos o nome da variável para referenciá-la
```

```
//PASSANDO FUNÇÃO COMO PARÂMETRO PARA UM ARRAY:
```

```
const array = [function (a, b) { return a + b; }, nome => `Ola ${nome}!`, soma]; //Perceba que passamos 3 funções de 3 formas diferente  
s: 1ª uma função literal para somar 2 valores, 2ª uma arrow function que retorna um nome passado e 3ª uma função que já havia sido de  
clarada mais acima...
```

```
console.log('\nPassando função dentro de array...')
```

```
console.log(array[0](2, 3));
```

```
console.log(array[1]('Gabriel'));
```

```
console.log(array[2](3, 3));

//PASSANDO FUNÇÃO COMO PARÂMETRO PARA UM OBJECT:
const obj = {boasVindas: () => 'Seja bem vindo!', NomeESobreNome: soma}; //Perceba que passamos 2 funções na inicialização do objetos
, onde: 1ª é uma arrow function sem assinatura e a 2ª é uma função declarada anteriormente...
obj.saudacao = function (nome) {return `Olá ${nome}`}; //Depois atribuímos uma função literalmente sobre uma chave "saudacao"...

console.log("\nFunções são passadas como parâmetro para o object...");
console.log(obj.saudacao('Gabriel'));
console.log(obj.boasVindas());
console.log('Seu nome completo é', obj.NomeESobreNome('Gabriel ', 'Ferreira Nobre'));

//PASSANDO FUNÇÃO COMO PARÂMETRO DE OUTRA FUNÇÃO:
function identificador(nome, valor1, valor2, funcao = soma) { //Perceba que a função identificador tem a missão de mostrar o nome e o
s valores que foram somados, enquanto recebe como padrão a função soma...
    return console.log(`${nome}, você somou os valores ${valor1} e ${valor2}, portanto o resultado da sua soma é ${funcao(valor1, val
or2)}`) //Veja que soma é executado aqui com o nome que recebeu como parâmetro "funcao"...
}

console.log("\nFunção soma é passada como parâmetro para uma função que adiciona um texto ao resultado...")
identificador('Gabriel', 2, 3); //Como a função já foi passada como parâmetro padrão, não precisamos passá-
la aqui na chamada, caso contrário teríamos que passá-la como parâmetro aqui...

//ARROW FUNCTION:
const somatoria = (a, b) => { //Perceba que a função arrow pode ser declarada usando o campo de parâmetros e a famosa seta do arrow, d
epois só precisamos abrir o corpo da função.
    return a + b;
}
console.log('\nFunção é uma arrow function...')
console.log(somatoria(15, 15)); //Como foi atribuída a uma variável, basta referenciá-la pelo nome da variável...
```

//ARROW RESUMIDA:

```
const adicao = (a, b) => a + b; //Perceba que a versão resumida da arrow function possui return implícito, não precisamos de declarar o seu return, porém, esse tipo de função resumida só funciona com funções que possui só uma única linha de código.  
console.log(`\nFunção é uma arrow function resumida...`)  
console.log(adicao(20, 20));
```

//ARROW VERSÃO AINDA MAIS RESUMIDA:

```
const nome = n => `Olá ${n}, tudo bem?`; //Quando a nossa função recebe um único parâmetro não precisamos usar os parênteses que delimitam o campo de parâmetros da função arrow.  
console.log(`\nFunção é uma arrow function ainda mais resumida...`)  
console.log(nome('Gabriel'));
```

//FUNÇÕES PODEM SER INVOCADAS COMO SE FOSSEM CLASSES:

```
const Cliente = function() {};  
console.log(`\nFunções podem ser declaradas como se fossem classes...`)  
console.log('1) ', typeof Cliente); //Perceba que o que temos realmente é uma função...  
console.log('2) ', typeof Cliente()); //podemos até chamá-la do modo convencional, vai gerar undefinel por não ter comportamento nenhum...  
console.log('3) ', typeof new Cliente); //porém, no Javascript podemos instanciá-la...
```

//A INFLUÊNCIA DO THIS SOBRE AS FUNCTION:

```
//Quando uma função é criada de forma normal usando "function" o this têm a característica de sempre referenciar ao objeto que o chamou...  
function quemEoThis() {  
    return console.log(  
        `Function comparada com o contexto Global: ${this === global}  
        Function comparada com o contexto Local da Chamada: ${this === this}`);  
}
```

```
console.log("\nFunction: Exemplo de this sendo chamado á partir de um contexto Global...")
quemEhOThis(); //Perceba que a chamada está no contexto global, o que significa que ele deverá dar true no primeiro exemplo e no segu
ndo...

//INFLUÊNCIA DAS ARROW FUNCTION SOBRE O THIS:
//Todo arrow function tem por característica ter o seu this dentro do contexto léxico, ou seja, onde a função foi criada, que no caso
é dentro de uma variável...
let sempreLocal = () => console.log( //Note que o this será referenciado dentro do contexto da variável let...
  `Arrow Function comparada com o contexto Global: ${this === global}
  Arrow Function comparada com o contexto Local da Chamada: ${this === this}`);
console.log("\nArrow Function: exemplo de this sendo chamado á partir de um contexto Global...")
sempreLocal(); //Perceba que estamos chamado pelo contexto global, ainda assim, ela vai referenciar ao contexto da variável, dando os
resultados "false" quando comparada ao contexto global e "true" quando comparada ao próprio contexto..."

//THIS DAS ARROW FUNCTION NÃO MUDA MESMO USANDO BIND OU APLICANDO A OUTRO CONTEXTO QUALQUER:
let comparaArrowFunction = (param) => console.log(this === param); //Perceba que criamos uma arrow function dentro de uma variável ch
amada "comparaArrowFunction" que recebe um contexto e compara com o this da arrow function retornando "true" para o this que for igua
l ao da arrow function e "false" para um this diferente...
console.log("\nArrow Function: Exemplos de comparação entre o this da arrow function e o this de outros contextos...")
console.log("Arrow function recebe \"global\" como parâmetro...");
comparaArrowFunction(global);
console.log("Arrow function recebe \"this\" como parâmetro...");
comparaArrowFunction(this);
console.log("Arrow function recebe \"module.exports\" como parâmetro...");
comparaArrowFunction(module.exports);
console.log("Arrow function é colocada dentro de um object...");
let objetinho = {};
objetinho.funcao = comparaArrowFunction;
console.log(objetinho);
console.log("Ainda assim, quando chamamos a arrow function comparando o this ao contexto do objeto, o resultado é...");
objetinho.funcao(objetinho);
```

```
//THIS EM ARRAYS:
//Quando um elemento está dentro de um array, ele passa automaticamente a fazer parte do contexto do array, ou seja, o seu this agora
é o array...
const euSouThisAgora = [quemEhOThis, sempreLocal];
console.log("\nTestando em Arrays:")
euSouThisAgora[0](); //Note como o padrão mudou para quando chamávamos uma função pelo contexto global, agora ela pertence ao context
o do array...
euSouThisAgora[1](); //Note que no arrow function não houve mudança, afinal as arrow function são leais aos contextos onde foram cria
das...


//THIS EM OBJECTS:
//Quando um elemento está dentro de um object, ele passa automaticamente a fazer parte do contexto do object, ou seja, o seu this ago
ra é o object...
const euSouThisNow = {quemEhOThis, sempreLocal};
console.log("\nTestando em Objects:")
euSouThisNow.quemEhOThis(); //Note como o padrão mudou para quando chamávamos uma função pelo contexto global, agora ela pertence ao
contexto do object...
euSouThisNow.sempreLocal(); //Note que no arrow function não houve mudança, afinal as arrow function são leais aos contextos onde for
am criadas...


//USANDO THIS PARA ACESSAR UM VALOR QUE ESTÁ DENTRO DE UM OBJETO:
const pessoa = {
  saudacao: 'Bom dia!',
  falar() { //Perceba que temos uma função dentro de um objeto que retorna um outro atributo daquele objeto por através da palavra
reservada "this"...
    return console.log(this.saudacao); //como this foi criado dentro de um object ele referencia ao object, perceba que trocamos
o nome do objeto pelo this naturalmente...
  }
}
console.log("\nUsando 'this' para chamar um atributo dentro do mesmo object:")
pessoa.falar();
```



```
//THIS NÃO PODE SER USADO PARA CHAMAR UM ATRIBUTO QUE PERTENCE A OUTRO OBJETO:
const outroObject = pessoa.falar; //Perceba que passamos para a constante a função falar do objeto pessoa...
console.log("\nFalhando em capturar o valor do atributo de outro objeto usando o this referenciado em outro objeto:")
outroObject(); //Porém quando chamamos a função ela executa e não consegue encontrar o valor da variável "saudacao", pois a chamada d
a função é um "this.saudacao" que está no contexto do objeto "pessoa", e não no contexto da constante "outroObject", por isso a chama
da da função gera um valor "undefined"...
```

```
//AMARRANDO A REFERÊNCIA DE UM ELEMENTO POR ATRAVÉS DO bind():
const usandoBindNoutroObject = pessoa.falar.bind(pessoa); //Perceba que fizemos a mesma operação de cima, tentando capturar o valor d
o atributo "saudacao" por através da chave "falar" do objeto "pessoa", porém, dessa vez usamos o bind() para amarrar a chamada da fun
ção "falar" ao objeto "pessoa"...
console.log("\nConseguindo pegar o valor do atributo de outro objeto que usa this referenciado a ele mesmo com a ajuda do bind():");
usandoBindNoutroObject(); //Veja como o resultado é diferente quando usamos "bind()"...
```

```
//FUNÇÕES LAZY:
//Funções lazy são funções que contém uma função dentro de si como retorno e que pedem por um valor específico adicional...
function operacaoMatematica(a, b){ //Essa função faz qualquer operação matemática com 2 valores...
    return function (c) { //Mas o usuário tem que especificar o tipo de função que ele deseja de 1 a 4...
        switch(c) {
            case 1: {
                console.log(a + b);
                break;
            }
            case 2: {
                console.log(a - b);
                break;
            }
            case 3: {
                console.log(a * b);
                break;
            }
            case 4: {
                console.log(a / b);
            }
        }
    }
}
```

```

        break;
    }
    default:
        console.log(a + b);
    }
}
}
console.log('\nFunção Lazy...');
operacaoMatematica(2, 3)(1); //Veja que a chamada de uma função lazy é totalmente diferente, pois incluímos um segundo campo de parâmetros que deverá ser satisfeito para que a função possa ocorrer corretamente...
operacaoMatematica(2, 3)(2);
operacaoMatematica(2, 3)(3);
operacaoMatematica(2, 3)(4);
operacaoMatematica(2, 3)(5);

```

RESULTADOS:

```
[Running] node "c:\Users\User\Documents\JAVASCRIPT\ARQUIVOS_DAS_AULAS\037-Manipulando_Functions.js"
```

Função recebe parâmetros mas não dá retorno...

5

5

undefined

Função não recebeu todos os parâmetros que deveria...

NaN

Função recebeu mais parâmetros do que deveria...

10

Função não recebe argumento nenhum...

NaN

Função recebe string no lugar de números e concatena eles...

Gabriel Ferreira

Quando a função não recebe parâmetro e mesmo assim passamos parâmetros na chamada da função e o recuperamos com arguments...

```
0
1
6.6
3.3000000000000003Teste
0abc
```

Função que utiliza retorno...

```
10
7
NaN
```

Quando funções retornam objetos...

```
6) Salario1 = true e Salario2 = true {
  comprarSorvete: true,
  comprarTV50: true,
  comprarTV32: false,
  manterSaudavel: false
}
```

```
7) Salario1 = true e Salario2 = false {
  comprarSorvete: true,
  comprarTV50: false,
  comprarTV32: true,
  manterSaudavel: false
}
```

```
8) Salario1 = false e Salario2 = true {
  comprarSorvete: true,
  comprarTV50: false,
  comprarTV32: true,
  manterSaudavel: false
}
```

```
9) Salario1 = false e Salario2 = false {
  comprarSorvete: false,
```

```
comprarTV50: false,  
comprarTV32: false,  
manterSaudavel: true  
}
```

Função que já possui valor padrão...

5

10

Quando desejamos declarar um valor padrão sobre uma variável declarada dentro de uma função...

A soma dos valores passados é: 3

A soma dos valores passados é: 6

Função é passada como parâmetro de uma variável...

20

Passando função dentro de array...

5

Ola Gabriel!

6

Funções são passadas como parâmetro para o object...

Olá Gabriel

Seja bem vindo!

Seu nome completo é Gabriel Ferreira Nobre

Função soma é passada como parâmetro para uma função que adiciona um texto ao resultado...

Gabriel, você somou os valores 2 e 3, portanto o resultado da sua soma é 5

Função é uma arrow function...

30

Função é uma arrow function resumida...

40

Função é uma arrow function ainda mais resumida...

Olá Gabriel, tudo bem?

Funções podem ser declaradas como se fossem classes...

- 1) function
- 2) undefined
- 3) object

Function: Exemplo de this sendo chamado á partir de um contexto Global...

Function comparada com o contexto Global: true

Function comparada com o contexto Local da Chamada: true

Arrow Function: exemplo de this sendo chamado á partir de um contexto Global...

Arrow Function comparada com o contexto Global: false

Arrow Function comparada com o contexto Local da Chamada: true

Arrow Function: Exemplos de comparação entre o this da arrow function e o this de outros contextos...

Arrow function recebe "global" como parâmetro...

false

Arrow function recebe "this" como parâmetro...

true

Arrow function recebe "module.exports" como parâmetro...

true

Arrow function é colocada dentro de um object...

```
{ funcao: [Function: comparaArrowFunction] }
```

Ainda assim, quando chamamos a arrow function comparando o this ao contexto do objeto, o resultado é...

false

Testando em Arrays:

Function comparada com o contexto Global: false

Function comparada com o contexto Local da Chamada: true

Arrow Function comparada com o contexto Global: false

Arrow Function comparada com o contexto Local da Chamada: true

Testando em Objects:

Function comparada com o contexto Global: false

Function comparada com o contexto Local da Chamada: true

```
Arrow Function comparada com o contexto Global: false  
Arrow Function comparada com o contexto Local da Chamada: true
```

```
Usando 'this' para chamar um atributo dentro do mesmo object:  
Bom dia!
```

```
Falhando em capturar o valor do atributo de outro objeto usando o this referenciado em outro objeto:  
undefined
```

```
Conseguindo pegar o valor do atributo de outro objeto que usa this referenciado a ele mesmo com a ajuda do bind():  
Bom dia!
```

```
Função Lazy...
```

```
5
```

```
-1
```

```
6
```

```
0.6666666666666666
```

```
5
```

```
[Done] exited with code=0 in 0.177 seconds
```