

Desenvolvimento BACK-END I

Professor: Douglas Legramante

E-mail: douglas.legramante@ifro.edu.br



INSTITUTO FEDERAL
Rondônia
Campus Vilhena

Function() {



JavaScript

(Aula 09) => {Funções}

}

Funções

De modo geral, função é um "subprograma" que pode ser chamado por código externo (ou interno no caso de recursão) à função. Assim como o programa em si, uma função é composta por uma sequência de instruções chamada corpo da função. Valores podem ser passados para uma função e ela vai retornar um valor.

Uma função é um conjunto de instruções que executa uma tarefa ou calcula um valor.

Por que são úteis?

O uso de funções permite o **reaproveitamento** de código.

Usar funções facilita a manutenção do código, pois qualquer alteração é feita em um único lugar. Sem funções precisamos fazer alterações em vários lugares, dificultando a manutenção da aplicação.

Declarando Funções

Atualmente existem cinco tipos de definições, sendo:


- *Functions declaration* (declaração de função)
- *Functions expression* (expressão de função)
- *Arrow Functions* (função de seta)
- *Functions constructor* (função construtora)

As definições mais comuns são *declaration* e *expression*.

Declaração de função

A definição da *function declaration* consiste no uso da palavra-chave **function**, seguida por:

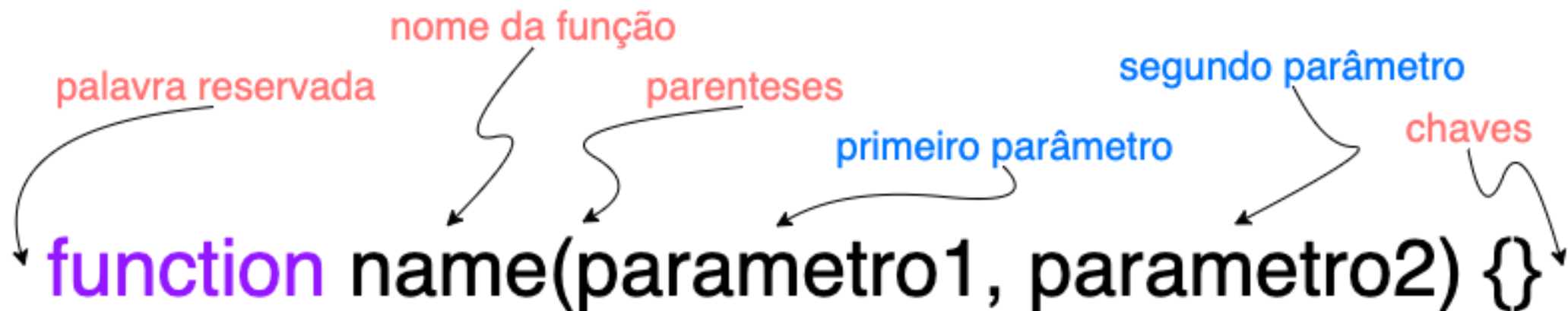
- Nome da Função.
- Lista de parâmetros para a função, entre parênteses e separados por vírgulas.
- Declarações JavaScript que definem o corpo da função, entre chaves { }.



The diagram illustrates the syntax of a function declaration: `function name() {}`. Red labels with arrows point to specific parts: 'palavra reservada' points to 'function', 'nome da função' points to 'name', 'parenteses' points to '()', and 'chaves' points to '{}'. The word 'function' is highlighted in purple.

Declaração de função

Também podemos definir parâmetros opcionais separados por vírgula:



The diagram shows the function declaration `function name(parametro1, parametro2) {}` with labels and arrows pointing to its components:

- palavra reservada** (red) points to `function` (purple).
- nome da função** (red) points to `name` (black).
- parenteses** (red) points to the opening parenthesis `(` (black).
- primeiro parâmetro** (blue) points to `parametro1` (black).
- segundo parâmetro** (blue) points to `parametro2` (black).
- chaves** (red) points to the closing curly brace `}` (black).

Exemplo

A função `square` recebe um argumento chamado `numero`.

```
function square(numero) {  
    return numero * numero;  
}
```

A declaração `return` especifica o valor retornado pela função.

Parâmetros primitivos (como um número) são passados para as funções por valor; o valor é passado para a função, mas se a função altera o valor do parâmetro, esta mudança não reflete globalmente.

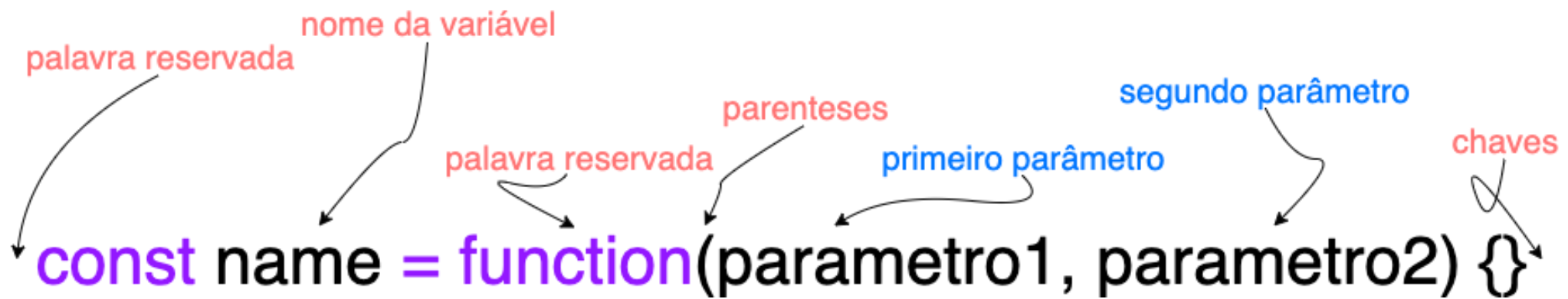
Exemplo

Se você passar um objeto (ou seja, um **valor não primitivo**, tal como **Array** ou um objeto definido por você) como um parâmetro e a função alterar as propriedades do objeto, essa mudança é visível fora da função

```
function minhaFuncao(objeto) {  
    objeto.make = "Toyota";  
}  
  
var meucarro = { make: "Honda", model: "Accord", year: 1998 };  
var x, y;  
  
x = meucarro.make; // x recebe o valor "Honda"  
  
minhaFuncao(meucarro);  
  
y = meucarro.make; // y recebe o valor "Toyota"  
// (a propriedade make foi alterada pela função)
```

Expressão de função

Uma *function expression* é muito semelhante e tem quase a mesma sintaxe de uma *function declaration*. A principal diferença entre uma *function expression* e uma *function declaration* é o nome da função, que pode ser omitido para criar funções anônimas.



The diagram illustrates the syntax of a function expression with the following components and labels:

- palavra reservada** (reserved word) pointing to `const`
- nome da variável** (variable name) pointing to `name`
- palavra reservada** (reserved word) pointing to `=`
- parenteses** (parentheses) pointing to `function`
- primeiro parâmetro** (first parameter) pointing to `parametro1`
- segundo parâmetro** (second parameter) pointing to `parametro2`
- chaves** (brackets) pointing to `{ }`

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

Expressão de função

Embora a declaração de função “square” do slide anterior seja sintaticamente uma declaração válida, funções também podem ser criadas por uma **expressão de função**. Tal função pode ser **anônima**; não tem que ter um nome. Por exemplo, a função **square** poderia ter sido definida como:

```
var square = function (numero) {  
    return numero * numero;  
};  
  
var x = square(4); //x recebe o valor 16
```

Expressão de função

No entanto, um nome pode ser fornecido com uma expressão de função e pode ser utilizado no interior da função para se referir a si mesma:

```
var fatorial = function fac(n) {  
    return n < 2 ? 1 : n * fac(n - 1);  
};  
  
console.log(fatorial(3));
```

As expressões de função são convenientes ao passar uma função como um argumento para outra função.

Expressão de função

Em JavaScript, uma função pode ser definida com base numa condição. Por exemplo, a seguinte definição de função define `minhaFuncao` somente se `num` é igual a 0:

```
var minhaFuncao;  
if (num == 0) {  
    minhaFuncao = function (objeto) {  
        objeto.make = "Toyota";  
    };  
}
```

Declarando Funções

■ Declaração de Função:

- As declarações de função são içadas (hoisting), o que significa que podem ser usadas antes de serem declaradas no código.
- São definidas com a palavra-chave `function`.

```
function somar(a, b) {  
  return a + b;  
}
```

■ Expressão de Função:

- As expressões de função não são içadas e só podem ser usadas após a sua declaração no código.
- São geralmente definidas como variáveis.

```
const subtrair = function(a, b) {  
  return a - b;  
};
```

Chamando funções

A definição de uma função não a executa. Definir a função é simplesmente nomear a função e especificar o que fazer quando a função é chamada. Chamar a função executa realmente as ações especificadas com os parâmetros indicados. Por exemplo, se você definir a função **square**, você pode chamá-la do seguinte modo:

```
function square(numero) {  
    return numero * numero;  
}
```

Declaração

```
square(5);
```

Chama a função com o argumento 5. A função executa as instruções e retorna o valor 25.

Escopo da função

As variáveis definidas no interior de uma função não podem ser acessadas de nenhum lugar fora da função, porque a variável está definida apenas no escopo da função. No entanto, uma função pode acessar todas variáveis e funções definida fora do escopo onde ela está definida. Em outras palavras, a função definida no escopo global pode acessar todas as variáveis definidas no escopo global. A função definida dentro de outra função também pode acessar todas as variáveis definidas na função hospedeira e outras variáveis ao qual a função hospedeira tem acesso.

Escopo da função

```
// As seguintes variáveis são definidas no escopo global
var num1 = 20,
    num2 = 3,
    nome = "Daniel";

// Esta função é definida no escopo global
function multiplica() {
    return num1 * num2;
}

multiplica(); // Retorna 60

// Um exemplo de função aninhada
function getScore() {
    var num1 = 2,
        num2 = 3;

    function add() {
        return nome + " marcou " + (num1 + num2) + " pontos";
    }
    return add();
}

getScore(); // Retorna "Daniel marcou 5 pontos"
```

Recursão

Uma função pode referir-se e chamar a si própria. Uma função que chama a si mesma é chamada de função recursiva. Em alguns casos, a recursividade é análoga a um laço. Ambos executam o código várias vezes, e ambos necessitam de uma condição (para evitar um laço infinito, ou melhor, recursão infinita, neste caso).

```
function loop(x) {  
  if (x >= 10)  
    // "x >= 10" a condição de parada (equivalente a "!(x < 10)")  
    return;  
  // faça coisas  
  loop(x + 1); // chamada recursiva  
}  
loop(0);
```

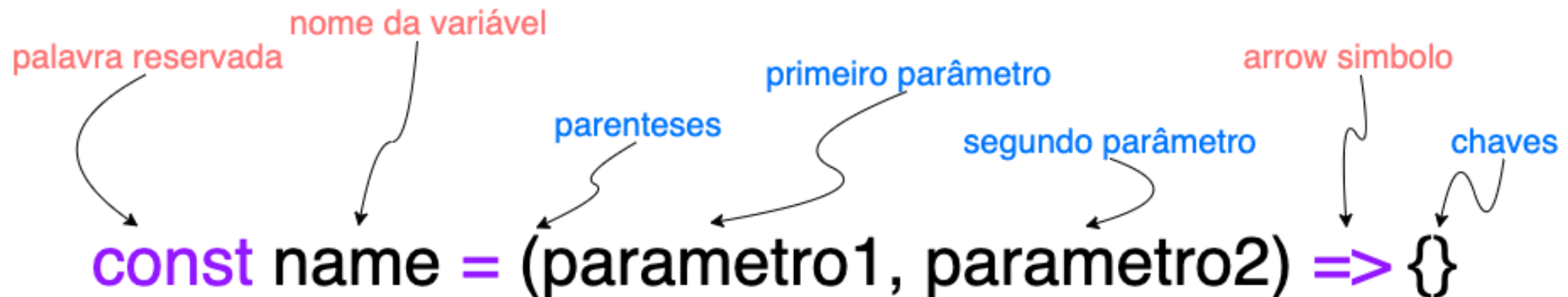
Funções de seta =>

As funções de seta são uma forma mais concisa de escrever funções em JavaScript, introduzidas no ES6 (ECMAScript 2015). Elas têm algumas diferenças importantes em comparação com as funções tradicionais.

- Funções de seta são sempre anônimas.
- São mais curtas e mais legíveis para funções simples.
- Não podem ser usadas como construtores.

Funções de seta =>

Em outras palavras, as *arrow functions* são simplificações para as *functions expression*.



The diagram illustrates the syntax of an arrow function with the following components and labels:

- palavra reservada** (reserved word) points to **const**.
- nome da variável** (variable name) points to **name**.
- parenteses** (parentheses) points to the opening parenthesis **(**.
- primeiro parâmetro** (first parameter) points to **parametro1**.
- segundo parâmetro** (second parameter) points to **parametro2**.
- arrow símbolo** (arrow symbol) points to the arrow **=>**.
- chaves** (braces) points to the closing curly brace **}**.

The code snippet shown is: **const name = (parametro1, parametro2) => {}**

Caso o corpo da *arrow function* tenha apenas uma linha, podemos omitir a declaração das chaves.

Funções de seta =>

Transformando expressão de função em função de seta.

Expressão de função:

```
const numeroAleatorio = function () {  
  return Math.random()  
}
```

Quando o corpo da expressão de função possuí apenas uma linha de código, podemos omitir as chaves e a palavra `return` e transformá-la em uma função de seta:

```
const numeroAleatorio = () => Math.random()
```

Outro exemplo:

Expressão de função:

```
function nomeCompleto(nome, sobrenome) {  
  return `${nome} ${sobrenome}`  
}
```

Função de seta:

```
const nomeCompleto = (nome, sobrenome) => `${nome} ${sobrenome}`
```

Função construtora

As funções construtoras são declaradas e definidas como qualquer outra *expression* ou *declaration*, a principal diferença está na maneira como ela é invocada.

As funções construtoras precisam ser invocadas com a palavra reservada **new**:

```
function Pessoa(nome) {  
    this.nome = nome  
}  
const p = new Pessoa('Matheus') // { nome: 'Matheus' }
```

Normalmente o nome de funções construtoras começa com a primeira letra maiúscula.

Funções pré-definidas

JavaScript tem várias
funções pré-definidas.

`console.log()`

Exibe mensagens de depuração no console do navegador ou Node.js.

`parseFloat()`

Converte uma string em um número de ponto flutuante.

`parseInt()`

Converte uma string em um número inteiro.

`String()`

Converte um valor em uma string.

`Number()`

Converte um valor em um número.

`isNaN()`

Determina se um valor é NaN ou não.

Consulte mais funções nativas em:

<https://www.dio.me/articles/lista-de-funcoes-nativas-do-javascript>

Exercícios

Nível Fácil

01. Crie uma função que receba uma string como parâmetro e retorne a mesma string com todas as letras em caixa alta.
02. Crie uma função que receba um número como parâmetro e verifique se ele é par ou ímpar. Retorne uma string.
03. Crie uma função que receba dois números como parâmetros e retorne a soma deles.
04. Crie uma função que receba um valor e uma porcentagem como parâmetros. A função deve retornar o valor acrescido da porcentagem indicada.
05. Desenvolva uma função que calcule o valor de um produto com desconto. A função deve receber o valor original do produto e o percentual de desconto como parâmetros, e retornar o valor com desconto aplicado. Utilize essa função para calcular o valor final de diferentes produtos.

Exercícios

Nível médio

06. Desenvolva uma função que calcule a área de um círculo. A função deve receber o raio como parâmetro e retornar a área calculada.

07. Desenvolva uma função que converta uma temperatura de Celsius para Fahrenheit. A função deve receber a temperatura em Celsius como parâmetro e retornar a temperatura convertida.

08. Crie uma função que calcula o IMC. A função deve receber a altura e o peso como parâmetros e retornar o IMC.

Exercícios

Nível desafiador

09. Crie uma função que valide se uma senha atende aos critérios estabelecidos, como ter no mínimo 8 caracteres, pelo menos uma letra maiúscula, pelo menos uma letra minúscula e pelo menos um caractere especial.

10. Desenvolva uma função chamada validarCPF que recebe um CPF como uma string numérica e verifica se ele é válido de acordo com as regras de validação de CPF. A função deve retornar true se o CPF for válido e false caso contrário.

11. Escreva uma função que informe o retorno de um investimento (montante) com base nos valores do capital inicial, tempo em meses e taxa de juros mensal, fornecidos pelo usuário.

Use a fórmula: $M = C * (1+i)^t$

Onde:

C = Capital inicial investido

i = Taxa de juros, em percentual

t = Tempo do investimento, em meses

Exiba o resultado com duas casas decimais.

Para saber mais

Funções

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Functions>

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Functions>

Definindo funções em JavaScript

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

Cursos Gratuitos

Curso JavaScript Curso em Video

<https://www.cursoemvideo.com/curso/javascript/>

Curso JavaScript YouTube

<https://www.youtube.com/watch?v=McKNP3g6VBA>