Funções em JavaScript

O que são funções?

Funções são partes muito importantes da programação.
Uma função é um **bloco de código que serve para executar uma tarefa específica**.

Você pode usar a mesma função várias vezes, sem precisar repetir o código.
A função só é executada quando ela é **chamada** (ou invocada).

Exemplo:
Função que calcula a multiplicação de dois números:

```
function myFunction(p1, p2) {
  return p1 * p2;
}
```

Sintaxe de uma função em JavaScript

A estrutura básica de uma função é:

```
function nome(p1, p2, ...) {
  // código que será executado
}
```

Uma função é criada usando a palavra-chave **function**, seguida por:

- O **nome da função**
- Parênteses **( )**, onde ficam os parâmetros
- Chaves **{ }**, onde fica o código

O nome da função segue as mesmas regras de nomes de variáveis.
Os **parâmetros** ficam dentro dos parênteses.
O **código da função** fica dentro das chaves.

Uma função pode **retornar um valor** para quem a chamou, usando `return`.

Por que usar funções?

As funções ajudam a deixar o código:

- Mais organizado
- Mais fácil de entender
- Mais eficiente

Com funções, você pode:

- Reutilizar o mesmo código várias vezes
- Usar valores diferentes (argumentos)
- Obter resultados diferentes com a mesma função

Chamando (invocando) uma função

O código dentro da função será executado quando a função for chamada:

- Por outro código JavaScript
- Por um evento (ex: clique em um botão)
- Automaticamente

Os **parênteses** `()` servem para chamar a função.

Exemplo:
Função que converte Fahrenheit para Celsius:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit - 32);
}

let value = toCelsius(77);
```

Parâmetros incorretos

Se você chamar a função sem passar o valor esperado, o resultado pode ser errado:

```
let value = toCelsius();
```

Função sem parênteses

Se você acessar a função **sem** `()`, o JavaScript retorna a própria função, e não o resultado:

```
let value = toCelsius;
```

Observação:

- `toCelsius` → é a função
- `toCelsius()` → é o resultado da função

Funções de seta (Arrow Functions)

As funções de seta foram criadas no **ES6** e servem para escrever funções de forma mais curta.

Forma tradicional:

```
let myFunction = function(a, b) {
  return a * b;
}
```

Com arrow function:

```
let myFunction = (a, b) => a * b;
```

Essa forma é mais simples e muito usada atualmente.

Variáveis locais

Uma variável criada **dentro de uma função** só existe dentro dela.
Ela é chamada de **variável local**.

Exemplo:

```
function myFunction() {
  let carName = "Volvo";
}
```

Fora da função, `carName` **não pode ser usada**.

Variáveis locais:

- Só funcionam dentro da função
- Podem ter o mesmo nome em funções diferentes
- São criadas quando a função começa
- São apagadas quando a função termina

Parâmetros e argumentos

Esses dois conceitos são diferentes:

- **Parâmetros**: nomes usados na definição da função
- **Argumentos**: valores passados quando a função é chamada

Exemplo:

```
function greet(name, age) {
  return `Hello ${name}! You are ${age} years old.`;
}
```

Aqui:

- `name` e `age` → parâmetros
- `"John"` e `21` → argumentos

```
greet("John", 21);
```

Funções usadas como variáveis

O valor retornado por uma função pode ser usado como se fosse uma variável.

Exemplo tradicional:

```
let x = toCelsius(77);
let text = "The temperature is " + x + " Celsius";
```

Usando a função diretamente:

```
let text = "The temperature is " + toCelsius(77) + " Celsius";
```

Isso deixa o código mais curto e simples.

Funções não fazem verificação automática

Em JavaScript, uma função **não verifica automaticamente**:

- O tipo dos valores recebidos
- A quantidade de valores passados

Ou seja, o JavaScript confia no que você envia para a função.

Parâmetros e argumentos da função

Uma função pode receber **parâmetros**, que são definidos quando a função é criada:

```
function functionName(parameter1, parameter2, parameter3) {
  // código a ser executado
}
```

- **Parâmetros** → são os nomes usados na definição da função
- **Argumentos** → são os valores reais passados para a função quando ela é chamada

---

Regras dos parâmetros em JavaScript

Em JavaScript:

- As funções **não definem tipos de dados** para os parâmetros
- As funções **não verificam o tipo** dos argumentos recebidos
- As funções **não verificam a quantidade** de argumentos passados

Isso significa que você pode passar menos ou mais valores do que o esperado, e o JavaScript não gera erro automaticamente.

Parâmetros padrão

Se uma função for chamada com **menos argumentos do que o declarado**, os parâmetros que faltam recebem o valor `undefined`.

Exemplo:

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 2;
  }
}
```

Nesse caso, se `y` não for passado, ele recebe o valor `2`.

Valores padrão de parâmetros (ES6)

A partir do ES6, é possível definir um **valor padrão diretamente no parâmetro**.

Exemplo:
Se `y` não for informado, ele terá o valor `10`.

```
function myFunction(x, y = 10) {
  return x + y;
}

myFunction(5);
```

Resultado: `15`

Parâmetro rest (...)

O **parâmetro rest** permite que uma função receba **quantos argumentos forem necessários**, tratando todos como uma lista (array).

Exemplo:

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) {
    sum += arg;
  }
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

Aqui, todos os valores passados são armazenados dentro de `args`.

Objeto **`arguments`**

Toda função JavaScript possui um objeto interno chamado **`arguments`**.

Esse objeto guarda **todos os argumentos usados quando a função foi chamada**, mesmo que a função não tenha parâmetros definidos.

Exemplo: encontrar o maior número

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
```

```
  let max = -Infinity;
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
```

Essa função percorre todos os valores recebidos e retorna o maior.

Exemplo: somar todos os valores

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

Essa função soma todos os argumentos passados, sem precisar definir parâmetros.

Expressões de Função em JavaScript

O que é uma expressão de função?

Uma **expressão de função** é uma função que é **armazenada dentro de uma variável**.

Em vez de criar a função separadamente, ela é definida **como parte de uma expressão**.

Uma expressão de função pode:

- Ser guardada em uma variável
- Ser passada como argumento para outra função
- Ser retornada por outra função

Exemplo de expressão de função

```
const x = function (a, b) {
  return a * b;
};
```

Nesse caso, a função foi atribuída à variável `x`.

Depois disso, a variável pode ser usada como se fosse a própria função:

```
let z = x(4, 3);
```

Função anônima

A função acima é uma **função anônima**, ou seja, **não tem nome**.

Quando uma função é armazenada em uma variável:

- Ela **não precisa ter nome**
- Ela é chamada usando o **nome da variável**

Observação importante

As expressões de função terminam com **ponto e vírgula (`;`)**, porque fazem parte de uma instrução executável:

```
const x = function (a, b) { return a * b; };
```

Por que expressões de função são importantes?

Expressões de função são muito usadas em JavaScript por serem flexíveis.
Você verá muito mais sobre elas adiante, especialmente em:

Callbacks

Funções passadas como argumentos para outras funções, muito comuns em:

- Eventos (cliques, teclado)
- Operações assíncronas

Closures

Permitem que uma função **lembre e use variáveis** do seu próprio escopo, mesmo depois de ser executada.

Funções de seta (Arrow Functions)

Uma forma mais curta e moderna de escrever expressões de função:

```
const add = (a, b) => a + b;
```

IIFEs (Funções Invocadas Imediatamente)

Funções que são executadas **assim que são criadas**, usadas para evitar conflitos com variáveis globais.

Declaração de Função vs Expressão de Função

Em JavaScript, existem **duas formas principais** de criar funções:

- **Declaração de função**
- **Expressão de função**

Elas funcionam de maneira diferente.

Declaração de função

Uma declaração de função usa a palavra-chave `function` e **exige um nome**:

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

Hoisting (elevação)

Declarações de função são **elevadas** para o topo do código.
Isso significa que você pode chamar a função **antes de ela ser definida**:

```
let sum = add(2, 3);

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

Observação

Declarações de função normalmente **não usam ponto e vírgula**, pois não são instruções executáveis.

Expressão de função

Uma expressão de função é criada quando uma função é atribuída a uma variável:

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

Ela geralmente é **anônima**, mas também pode ter nome:

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

Sem hoisting

Expressões de função **não são elevadas** como declarações de função.
Você **não pode chamá-las antes de serem definidas**:

```
let sum = add(2, 3); // ✕ Erro

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

Principais diferenças

Declaração de função

- Precisa ter nome
- É elevada (hoisting)
- Pode ser chamada antes da definição
- Indicada para funções de uso geral

Expressão de função

- Pode ser anônima
- Não é elevada
- Só pode ser usada após ser definida
- Muito usada em callbacks, eventos e código moderno

Resumo em tabela

Declaração de Função

- Usa `function` com nome obrigatório
- Pode ser chamada antes da definição
- Ideal para funções principais do sistema

Expressão de Função

- Faz parte de uma atribuição
- Pode ser anônima
- Só funciona após ser definida
- Muito flexível e moderna