



# Funções e Condicionais em JavaScript

☰ Conhecimentos	2. Plataformas de desenvolvimento: conceitos. Tipos. Características e especificações técnicas. 4. Lógica de Programação: Conceito de algoritmo. Algoritmos naturais e estruturados. Representação
📅 Data da aula	@22 de maio de 2024
☰ Tipo	Atividade em Classe   Aula Expositiva



**Serviço Nacional de Aprendizagem Comercial**

**Departamento Regional de Pernambuco**

**Habilitação Profissional Técnica em Informática**

**UC9 - Desenvolver Algoritmos**

**Thiago Nogueira - Instrutor de Educação Tecnológica**

✉ thiago.nogueira@senac.pe.br

🌐 linkedin.com/tdn

☎ (81) 9 9627-0419

📷 @thiagoo.\_nogueiraa

## Objetivos

1. Aprender sobre Declarações Condicionais
2. Aprender sobre operadores de comparação e lógicos
3. Aprender sobre declarar funções
4. Aprender sobre expressar funções

## Declarações Condicionais

### Declaração If

Na programação, podemos realizar uma tarefa baseada em uma condição usando uma instrução `if`:

```
if (true) {
  console.log('Essa mensagem será impressa!');
}
// Imprime: Essa mensagem será impressa!
```

A `if` declaração é composta por:

- A palavra-chave `if` seguida por um conjunto de parênteses `()` que é seguido por um *bloco de código*, indicado por um conjunto de chaves `{}`.
- Dentro dos parênteses `()`, é fornecida uma condição avaliada como `true` ou `false`.
- Se a condição for avaliada como `true`, o código entre chaves `{}` será executado.
- Se a condição for avaliada como `false`, o bloco não será executado.

## Declarações If...Else

Em muitos casos, teremos código que queremos executar se nossa condição for avaliada como `false`. Se quisermos adicionar algum comportamento padrão a instrução `if`, podemos adicionar uma instrução `else` para executar um bloco de código quando a condição for avaliada como `false`.

```
if (false) {
  console.log('Esse bloco de código não será executado.');
```

```
} else {
  console.log('Mas esse bloco será!');
```

```
}
```

```
// Imprime: Mas esse bloco será!
```

Uma declaração `else` deve ser utilizada com uma declaração `if` e, juntas, elas são chamadas de declaração `if...else`.

No exemplo acima, a declaração `else`:

- Usa a palavra-chave `else` após o bloco de código de uma declaração `if`.
- Possui um bloco de código envolvido por um conjunto de chaves `{}`.
- O código dentro do bloco da instrução `else` será executado quando a condição `if` da instrução for avaliada como `false`.

## Operadores de comparação

Ao escrever instruções condicionais, às vezes precisamos usar diferentes tipos de operadores para comparar valores. Esses operadores são chamados de *operadores de comparação*.

Lista de alguns operadores de comparação e sua sintaxe:

- Menor que: `<`
- Maior que: `>`
- Menos que ou igual a: `<=`
- Melhor que ou igual a: `>=`
- É igual a: `===`
- Não é igual a: `!==`
- OBS: `==` e `===` são diferentes. Uma **comparação estrita** (`===`) só é verdade se os operandos são do mesmo tipo e possuem o mesmo valor. A **comparação** mais usada é a abstrata (`==`), que converte os operandos para o mesmo tipo antes de fazer a **comparação**.

## Operadores lógicos

Trabalhar com [condicionais](#) significa que usaremos booleanos `true` ou `false` valores. Em JavaScript, existem [operadores](#) que trabalham com valores booleanos conhecidos como *operadores lógicos*. Podemos usar

operadores lógicos para adicionar lógica mais sofisticada às nossas condicionais. Existem três operadores lógicos:

- o operador *and* (`&&`)
- o operador *or* (`||`)
- o operador *not* (`!`)

Quando usamos o operador `&&`, estamos verificando se duas coisas são `true`:

```
if (stopLight === 'green' && pedestrians === 0) {
  console.log('Go!');
} else {
  console.log('Stop');
}
```

Ao usar o operador `&&`, ambas as condições *devem ser* avaliadas como `true` para que toda a condição seja avaliada `true` e executada. Caso contrário, se qualquer uma das condições for `false`, a condição `&&` será avaliada como `false` e o bloco `else` será executado.

Se só uma condição `true` for o suficiente para tornar toda a sentença `true`, pode-se utilizar o operador `||`:

```
if (day === 'Saturday' || day === 'Sunday') {
  console.log('Enjoy the weekend!');
} else {
  console.log('Do some work.');
```

O `!` operador *not* inverte ou *nega* o valor de um booleano:

```
let excited = true;
console.log(!excited); // Imprime false

let sleepy = false;
console.log(!sleepy); // Imprime true
```

Operadores lógicos são frequentemente usados em [instruções](#) condicionais para adicionar outra camada de lógica ao nosso código.

## Operadores short-circuit

Digamos que você tenha um site e queira usar o nome de usuário de um usuário para fazer uma saudação personalizada. Às vezes, o usuário não possui conta, tornando a `username` variável falsa. O código abaixo verifica se `username` está definido e atribui uma string padrão caso não esteja:

```
let username = '';
let defaultName;

if (username) {
  defaultName = username;
} else {
  defaultName = 'Stranger';
}

console.log(defaultName); // Imprime: Stranger
```

Pode-se usar uma abreviação para o código acima. Em uma condição booleana, o JavaScript atribui o valor verdadeiro a uma variável se você usar o operador `||` em sua atribuição:

```
let username = '';
let defaultName = username || 'Stranger';
```

```
console.log(defaultName); // Imprime: Stranger
```

## Operador ternário

Com o espírito de usar sintaxe abreviada, podemos usar um *operador ternário* para simplificar uma instrução `if...else`. Por exemplo:

```
let eNoite = true;

if (eNoite) {
  console.log('Ligue as luzes!');
} else {
  console.log('Desligue as luzes!');
}
```

Podemos usar um *operador ternário* para realizar a mesma funcionalidade:

```
eNoite ? console.log('Ligue as luzes!') : console.log('Desligue as luzes!');
```

No exemplo acima:

- A condição, `eNoite`, é fornecida antes do `?`.
- As duas expressões seguintes, depois do `?`, são separadas por dois pontos `:`.
- Se a condição for avaliada como `true`, a primeira expressão será executada.
- Se a condição for avaliada como `false`, a segunda expressão será executada.

## Declarações if else

Podemos adicionar mais condições ao nosso programa com uma declaração `else if`. A declaração `if else` permite mais de dois resultados possíveis. Você pode adicionar instruções desejadas para criar condicionais mais complexas!

A declaração `else if` sempre vem depois da `if` antes da `else`. A `else if` também requer uma condição. Vamos dar uma olhada na sintaxe:

```
let semaforo = 'amarelo';

if (semaforo === 'vermelho') {
  console.log('Pare!');
} else if (semaforo === 'amarelo') {
  console.log('Devegar!');
} else if (semaforo === 'verde') {
  console.log('Prosseguir!');
} else {
  console.log('Cuidado, desconhecido!');
}
```

As declarações `if else` permitem que você tenha vários resultados possíveis. Instruções `if // else if` e `else` são lidas de cima para baixo, então a primeira condição avaliada `true` de cima para baixo é o bloco que é executado.

## Declarações switch case

Na programação, muitas vezes precisamos verificar vários valores e lidar com cada um deles de maneira diferente. Utilizar `if else` em código funciona bem, mas imagine se precisássemos verificar 100 valores diferentes! Ter que escrever tantas declarações parece ser muito trabalhoso!

Uma instrução `switch` fornece uma sintaxe alternativa que é mais fácil de ler e escrever. A declaração é como se segue:

```
let item = 'papaia';

switch (item) {
  case 'tomate':
    console.log('Tomates custam R$0.49');
    break;
  case 'limão':
    console.log('Limões custam R$1.49');
    break;
  case 'papaia':
    console.log('Papias custam R$1.29');
    break;
  default:
    console.log('Item inválido');
    break;
}
```

- A palavra-chave `switch` inicia a instrução e é seguida por `( ... )`, que contém o valor que cada um `case` irá comparar. No exemplo, o valor ou expressão da instrução `switch` é `item`.
- Se o valor de `item` for `'tomate'`, o `case` seria executado com o bloco de código `console.log()`.
- O valor de `item` é `'papaia'`, então o terceiro `case` é executado e o texto `Papias custam $1.29` é registrado no console.
- A palavra-chave `break` diz ao computador para sair do bloco e não executar mais nenhum código ou verificar quaisquer outros casos dentro do bloco de código. Observação: sem `break`, o primeiro caso correspondente será executado, mas o mesmo acontecerá com todos os casos subsequentes, independentemente de corresponderem ou não — incluindo o padrão. Esse comportamento é diferente das instruções condicionais `if` / `else` que executam apenas um bloco de código.
- No final de cada declaração `switch`, há uma declaração `default`. Se nenhum dos `cases` for verdadeiro, o código da instrução `default` será executado.

## Atividades de Aprendizagem

### Exercício 1

Crie uma variável `venda` do tipo booleano. Utilize uma estrutura de condição para verificar o valor da variável `venda` e em caso positivo imprima na tela: "Hora de comprar!!" Caso negativo, imprima "Espere pela temporada de vendas."

### Exercício 2

Crie uma variável `fome` do tipo `number`. Crie uma estrutura que verifica o valor da variável criada e se o valor `>7` imprima na tela "Hora de comer!". Caso contrário, imprima "Comer depois!!"

### Exercício 3

Crie uma variável `isLocked` do tipo booleano. Utilizando operadores ternários, verifique se o valor da variável é verdadeira e em caso positivo imprima na tela "Você precisa destrancar a porta", caso contrário imprima ("A porta está destrancada.")

### Exercício 4

Utilizando uma estrutura do tipo `switch case`, crie um script que verifica o valor da posição de um atleta. Caso seja primeiro lugar, imprima na tela ("Você ganhou a medalha de ouro"), segundo lugar prata, terceiro lugar bronze e em outros casos imprima "Você não ganhou medalhas".

### Exercício 5

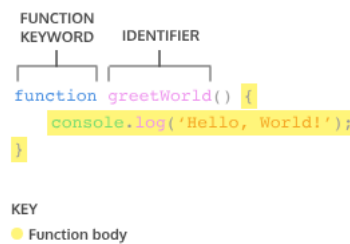
- Crie uma variável chamada `"diaSemana"` e atribua a ela um valor de 1 a 7, representando um dia da semana.
- Use uma estrutura `switch` para imprimir o nome do dia da semana correspondente ao valor da variável.
- Por exemplo, se `"diaSemana"` for igual a 1, imprima "Domingo" no console.

# Funções

Na programação, costumamos usar código para executar uma tarefa específica várias vezes. Em vez de reescrever o mesmo código, podemos agrupar um bloco de código e associá-lo a uma tarefa, e então podemos reutilizar esse bloco de código sempre que precisarmos executar a tarefa novamente. Conseguimos isso criando uma *função*. Uma função é um bloco de código reutilizável que agrupa uma sequência de [instruções](#) para executar uma tarefa específica.

## Declaração

Em JavaScript, existem muitas maneiras de criar uma [função](#). Uma maneira de criar uma função é usando uma *declaração*. Assim como uma declaração de variável vincula um valor a um nome de variável, uma declaração de função vincula uma função a um nome ou *identificador*.



Anatomia de uma declaração de função

Uma declaração de função consiste em:

- A palavra-chave `function`
- O nome da função, ou seu identificador, seguido de parênteses.
- Um corpo de função, ou o bloco de instruções necessárias para executar uma tarefa específica, entre colchetes da função, `{ }`.

Uma declaração de função é uma função vinculada a um identificador ou nome. Também devemos estar cientes do recurso de *hoisting* em JavaScript, que permite acesso às declarações de funções antes de serem definidas. Exemplo:

```
greetWorld(); // Output: Hello, World!

function greetWorld() {
  console.log('Hello, World!');
}
```

Observe que, por hoisting, pode-se chamar a função `greetWorld()` antes dela ser declarada.

## Chamando uma função

Como visto, uma declaração de função vincula uma função a um identificador. No entanto, uma declaração de função não solicita a execução do código dentro do corpo da função, apenas declara a existência da função. O código dentro do corpo de uma função é executado, somente quando a função é *chamada*.

Para [chamar uma função](#) em seu código, digite o nome da função seguido de parênteses.



Chamando uma função

Esta *chamada de função* executa o corpo da função ou todas as instruções entre chaves na declaração da função.

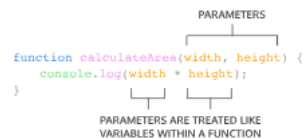


Fluxo de execução de uma função

Pode-se chamar a mesma função quantas vezes forem necessárias.

## Parâmetros e argumentos

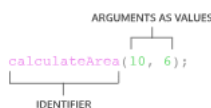
Até agora, as funções que criamos executam uma tarefa sem entrada. No entanto, algumas funções podem receber entradas e usá-las para executar uma tarefa. Ao declarar uma função, podemos especificar seus *parâmetros*. Os parâmetros permitem que as funções aceitem entradas e executem uma tarefa usando as entradas. Usamos parâmetros como espaços reservados para informações que serão passadas para a função quando ela for chamada.



Especificando parâmetros de uma função

No diagrama acima, `calculateArea()` calcula a área de um retângulo, com base em duas entradas, `width` e `height`. Os parâmetros são especificados entre parênteses como `width` e `height` e dentro do corpo da função, eles agem como variáveis regulares. `width` e `height` atuam como espaços reservados para valores que serão multiplicados juntos.

Ao chamar uma função que possui parâmetros, especificamos os valores entre parênteses que seguem o nome da função. Os valores que são passados para a função quando ela é chamada são chamados *de argumentos*.



Argumentos de uma função

Na chamada de função acima, o número `10` é passado como `width` e `6` é passado como `height`. Observe que a ordem na qual os argumentos são passados e atribuídos segue a ordem em que os parâmetros são declarados.



As variáveis `rectWidth` e `rectHeight` são inicializadas com os valores de altura e largura de um retângulo antes de serem usadas na chamada de função. Ao usar parâmetros, `calculateArea()` pode ser reutilizado para calcular a área de qualquer retângulo!

## Parâmetros Padrões

Um dos recursos adicionados no ES6 é a capacidade de usar *parâmetro padrão*. Eles permitem que os parâmetros tenham um valor predeterminado caso não haja nenhum argumento passado para a função ou se o argumento

for `undefined` chamado. Exemplo:

```
function greeting (name = 'estranho') {  
  console.log(`Olá, ${name}!`)  
}  
  
greeting('Nick') // Output: Olá, Nick!  
greeting() // Output: Olá, estranho!
```

- No exemplo acima, usamos o operador `=` para atribuir ao parâmetro `name` um valor padrão de `'estranho'`. Isso é útil caso queiramos incluir uma saudação padrão não personalizada!
- Quando o código chama `greeting('Nick')` o valor do argumento é passado e, `'Nick'`, substituirá o parâmetro padrão para `'estranho'` registrar `'Hello, Nick!'` no console.
- Quando não há um argumento passado para `greeting()`, o valor padrão de `'stranger'` é usado e `'Hello, stranger!'` registrado no console.

## Return


Quando uma função é chamada, o computador percorre o código da função e avalia o resultado. Por padrão, o valor resultante é `undefined`.

```
function rectangleArea(width, height) {  
  let area = width * height;  
}  
console.log(rectangleArea(5, 7)) // Imprime undefined
```

No exemplo de código, definimos a função para calcular o parâmetro `area`. A função é chamada com os argumentos mas ao imprimir os valores, obteve-se `undefined`. O que ocorreu?

A função funcionou bem e o computador calculou a área, mas não a capturamos. Então, como podemos fazer isso? Com a palavra-chave `return`!

```
function calculateArea(width, height) {  
  const area = width * height;  
  return area;  
}
```



RETURN KEYWORD      RETURN VALUE

Para retornar informações da chamada de função, usamos uma [instrução return](#). Para criar uma instrução de retorno, usamos a palavra-chave `return` seguida do valor que desejamos retornar. Como vimos acima, se o valor for omitido, `undefined` será retornado.

Quando uma instrução `return` é usada no corpo de uma função, a execução da função é interrompida e o código que a segue não será executado. Veja o exemplo:

```
function rectangleArea(width, height) {  
  if (width < 0 || height < 0) {  
    return 'You need positive integers to calculate area!';  
  }  
  return width * height;  
}
```

Se um argumento para `width` ou `height` for menor que `0`, então `rectangleArea()` retornará `'You need positive integers to calculate area!'`. A segunda instrução `return width * height` não será executada.

## Funções auxiliares



Também podemos usar o valor de retorno de uma função dentro de outra função. Essas funções chamadas dentro de outra função são frequentemente chamadas de *funções auxiliares*. Como cada função executa uma tarefa específica, nosso código fica mais fácil de ler e depurar, se necessário.

Se quiséssemos definir uma função que converta a temperatura de Celsius para Fahrenheit, poderíamos escrever duas funções como:

```
function multiplicaPorNoveQuintos(number) {  
  return number * (9/5);  
};  
  
function getFahrenheit(celsius) {  
  return multiplicaPorNoveQuintos(celsius) + 32;  
};  
  
getFahrenheit(15); // Retorna 59
```

No exemplo acima:

- `getFahrenheit()` é chamado e `15` passado como um argumento.
- O bloco de código dentro de `getFahrenheit()` chama `multiplicaPorNoveQuintos()` e passa `15` como argumento.
- `multiplicaPorNoveQuintos()` leva o argumento de `15` para o parâmetro `number`.
- O bloco de código dentro da `multiplicaPorNoveQuintos()` função multiplica `15` por `(9/5)`, que é avaliado como `27`.
- `27` é retornado para a chamada de função em `getFahrenheit()`.
- `getFahrenheit()` continua a ser executado. Ele adiciona `32` a `27`, que é avaliado como `59`.
- Finalmente, `59` é retornado para a chamada de função `getFahrenheit(15)`.

Podemos usar funções para separar pequenos pedaços de lógica ou tarefas e usá-las quando necessário. Escrever funções auxiliares pode ajudar a realizar tarefas grandes e difíceis e dividi-las em tarefas menores e mais gerenciáveis.

## Expressão de Função

Outra maneira de definir uma função é usar uma *expressão de função*. Para definir uma função dentro de uma expressão, podemos usar a palavra-chave `function`.

Em uma expressão de função, o nome da função geralmente é omitido. Uma função sem nome é chamada de *função anônima*. Uma expressão de função geralmente é armazenada em uma variável para fazer referência a ela.

Considere a seguinte expressão de função:

```
const calculateArea = function(width, height) {  
  const area = width * height;  
  return area;  
};
```

Características: Expressão de Função

Para declarar uma expressão de função:

1. Declare uma variável para fazer com que o nome da variável seja o nome da sua função.
2. Atribua como valor dessa variável uma função anônima criada usando a palavra `function` seguida por um conjunto de parênteses com parâmetros possíveis. Em seguida, um conjunto de chaves que contém o corpo da função.

Para invocar uma expressão de função, escreva o nome da variável na qual a função está armazenada, seguido de parênteses envolvendo todos os argumentos que estão sendo passados para a função.

```
variableName(argument1, argument2)
```

Ao contrário das declarações de função, as expressões de função não são elevadas, portanto não podem ser chamadas antes de serem definidas.

## Arrow Function

ES6 introduziu a *sintaxe de função de seta*, uma maneira mais curta de escrever funções usando a `() =>` notação especial “seta gorda”.

As funções de seta eliminam a necessidade de digitar a palavra-chave `function` sempre que você precisar criar uma função. Em vez disso, você primeiro inclui os parâmetros dentro de `()` e depois adiciona uma seta `=>` que aponta para o corpo da função cercado `{ }` assim:

```
const rectangleArea = (width, height) => {  
  let area = width * height;  
  return area;  
};
```

É importante estar familiarizado com as diversas maneiras de escrever funções porque você encontrará cada uma delas ao ler outro código JavaScript.

## Funções concisas

JavaScript também fornece várias maneiras de refatorar a sintaxe da função de seta. A forma mais condensada da função é conhecida como *corpo conciso*. Exploraremos algumas dessas técnicas abaixo:

1. Funções que usam apenas um único parâmetro não precisam que esse parâmetro esteja entre parênteses. No entanto, se uma função tiver zero ou vários parâmetros, serão necessários parênteses.

```
ZERO PARAMETERS  
const functionName = () => {};  
  
ONE PARAMETER  
const functionName = paramOne => {};  
  
TWO OR MORE PARAMETERS  
const functionName = (paramOne, paramTwo) => {};
```

2. Um corpo de função composto por um bloco de linha única não precisa de chaves. Sem as chaves, tudo o que essa linha avaliar será retornado automaticamente. O conteúdo do bloco deve seguir imediatamente a seta `=>` e a palavra `return` pode ser removida. Isso é conhecido como *retorno implícito*.
- 3.

```
SINGLE-LINE BLOCK  
const sumNumbers = number => number + number;  
  
MULTI-LINE BLOCK  
const sumNumbers = number => {  
  const sum = number + number;  
  return sum; } — RETURN STATEMENT  
};
```

Então, se tivermos uma função:

```
const squareNum = (num) => {  
  return num * num;  
};
```

Podemos refatorar a função para:

```
const squareNum = num => num * num;
```

## Atividades de Aprendizagem

### Exercício 1

Desenvolva uma função que receba um número como parâmetro e verifique se ele é par ou ímpar. Retorne true se for par e false se for ímpar.

### Exercício 2

Crie uma função que receba uma string como parâmetro e retorne a mesma string com todas as letras em caixa alta. Utilize essa função para converter diferentes strings.

### Exercício 3

Desenvolva uma função que determine se um número é primo ou não. Retorne true se for primo e false se não for.

### Exercício 4

Crie uma função que receba um valor e uma porcentagem como parâmetros. A função deve retornar o valor acrescido da porcentagem indicada.