Aula 2 - Sintaxe da Linguagem JavaScript

aTip Learn - Lógica de Programação com JavaScript

Objetivos da Aula

- Conhecer a sintaxe da linguagem JavaScript.
- Conhecer diferentes formas de declarar variáveis.
- Conhecer os principais operadores da linguagem JavaScript.
- Utilizar estruturas de controle if, else e elseif.
- Criar e utilizar funções em JavaScript.

Sintaxe da Linguagem JavaScript

O que é sintaxe?

Uma linguagem de programação, assim como uma linguagem natural, possui um conjunto de regras que determina como as palavras e símbolos devem ser organizados para formar frases e expressões que carreguem algum sentido ou significado. A essas regras damos o nome de sintaxe, ou regras sintáticas.

Enquanto na linguagem natural podemos nos expressar de diferentes formas, nas linguagens de programação temos um conjunto de regras mais rígidas que precisam ser seguidas para que o código seja interpretado corretamente pelo computador.

Comentários

Comentários são partes do código-fonte que são ignoradas pelo interpretador da linguagem. Dessa forma os comentários são geralmente utilizados para documentar o código explicando o que ele faz, o motivo de determinadas decisões, ou qualquer outra informação relevante para quem precisar ler e entender o código no futuro.

A linguagem JavaScript possui dois tipos de comentários:

1. **Comentários de Linha Única**: São comentários que ocupam apenas uma linha do código e são iniciados com `//`. Tudo que vem após `//` na mesma linha é considerado um comentário e é ignorado pelo interpretador.

```
// Este é um comentário de linha única
console.log('Olá Mundo!'); // O código no início da linha é executado, porém tudo que vier depois do //
```

2. **Comentários de Múltiplas Linhas**: São comentários que podem ocupar várias linhas do código e são iniciados com `/*` e finalizados com `*/`. Tudo que estiver entre `/*` e `*/` é considerado um comentário e é ignorado pelo interpretador.

```
console.log('Essa mensagem é exibida!');
/*
Este é um comentário de múltiplas linhas
Tudo que está entre os delimitadores é ignorado pelo interpretador
*/
console.log('Essa mensagem também é exibida!');
```

- Documentação sobre Comentários

Ponto e Vírgula

O ponto e vírgula é utilizado para separar instruções em JavaScript, e geralmente é utilizado no final de cada linha de código. Utilizar o ponto e vírgula no final de cada linha é uma boa prática da linguagem, porém, seu uso não é obrigatório, pois o interpretador da linguagem JavaScript é capaz de inferir o final de uma instrução mesmo sem o uso do ponto e vírgula. Entretanto, existem casos em que a ausência do ponto e

vírgula pode gerar erros de interpretação, além de dificultar o entendimento do código, por isso seu uso é recomendado e visto como uma boa prática.

```
let idade = 30 // Instrução sem ponto e vírgula
let nome = 'João'; // Instrução com ponto e vírgula
```

- Documentação sobre Ponto e Vírgula

Letras Maiúsculas e Minúsculas

JavaScript é uma linguagem case-sensitive, ou seja, ela faz distinção entre letras maiúsculas e minúsculas. Isso quer dizer que `nome`, `Nome` e `NOME` são considerados identificadores diferentes em JavaScript. Por conta disso, é importante manter a consistência na escrita do código, utilizando sempre o mesmo padrão de escrita para os identificadores.

```
let nome = 'João'; // Variável nome
let Nome = 'Maria'; // Variável Nome
let NOME = 'José'; // Variável NOME
```

Apesar de não existir uma regra para a escolha do padrão de escrita, a maioria dos programadores que utilizam a linguagem JavaScript adotam o padrão camelCase para nomear variáveis e funções. No padrão camelCase, a primeira palavra é escrita com letra minúscula, e as palavras seguintes são escritas com a primeira letra em maiúscula, sem espaços ou caracteres especiais entre as palavras.

```
let nomeCompleto = 'João Silva'; // Variável nomeCompleto
function exibirMensagem() { // Função exibirMensagem
    ...
}
```

Os programados de JavaScript também fazem uso do padrão PascalCase para nomear classes (veremos mais adiante o que são classes). No padrão PascalCase, todas as palavras são escritas com a primeira letra em maiúscula, sem espaços ou caracteres especiais entre as palavras.

```
class Usuario { // Classe Usuario
    ...
}
```

- Documentação sobre Convenções de Estilo (em Inglês)

Indentação

Indentação é a prática de adicionar espaços ou tabulações no início de uma linha de código para tornar o código mais legível. A indentação é utilizada para organizar o código, facilitando a leitura e entendimento do código, além de destacar a estrutura do código, como blocos de código, funções, estruturas de controle, etc. Algumas linguagens, como Python, utilizam a tabulação para definir blocos de código, enquanto outras, como JavaScript, utilizam a tabulação apenas para fins de organização e legibilidade, ou seja, escrever um código sem indentação não irá gerar um erro na linguagem, porém, irá tornar seu código mais díficil de ler e entender.

Convencionou-se que cada novo bloco de código deve ser indentado com um número fixo de espaços ou tabulações, geralmente 2 ou 4 espaços. A escolha do número de espaços é uma questão de preferência pessoal, ou da equipe de desenvolvimento, porém, é importante manter a consistência na escrita do código, utilizando sempre o mesmo número de espaços para indentar o código. A maioria dos editores de código modernos possuem funcionalidades que facilitam a indentação do código, identando automaticamente o código à medida que você escreve. Além disso existem ferramentas, como o utilitário Prettier, que são capazes de formatar o código automaticamente, seguindo um padrão de indentação pré-definido.

```
if (condicao) {
    // Bloco de código indentado com 4 espaços
    console.log('Primeira Mensagem');
```

```
    if (outraCondicao) {
        // Bloco de código indentado com 4 espaços
        console.log('Segunda Mensagem');
    }
}
```

Palavras-Chave

As palavras-chave são termos reservados da linguagem JavaScript que possuem um significado específico e não podem ser utilizados como nomes de variáveis, funções ou qualquer outro identificador da linguagem, são exemplos de palavras-chave `var`, `let` e `const` que utilizamos para declarar variáveis.

Você pode encontrar uma lista completa de palavras-chave da linguagem JavaScript no site da Mozilla Developer Network.

Declaração de Variáveis

Como vimos na aula anterior, as variáveis são utilizadas para identificar e referenciar dados armazenados na memória do computador, quando declaramos uma variável estamos dando um nome para um determinado espaço na memória do computador, podemos então usar esse nome para armazenar ou ler informações da memória.

A linguagem JavaScript permite que declaremos variáveis de três formas diferentes, cada forma é identificada por uma palavra-chave da linguagem:

1. **var**: É a forma mais antiga de declarar variáveis em JavaScript. Com a palavra-chave `var` criamos variáveis que podem ser lidas em qualquer parte do código, até mesmo antes da sua declaração, ou seja, o JavaScript permite que você utilize a variável em linhas de código que estão antes da sua declaração, uma funcionalidade conhecida como **hoisting**. Apesar de facilitar a escrita do código, o hoisting também pode ser motivo de confusão, e por conta disso nas versões mais recentes do JavaScript esse tipo de declaração caiu em desuso em favor das próximas duas formas de declaração.
2. **let**: A palavra-chave `let` foi introduzida no JavaScript a partir da versão ES6 (ECMAScript 2015) e é a forma mais recomendada de declarar variáveis em JavaScript. A declaração de variáveis com `let` não permite o hoisting, ou seja, você não pode utilizar a variável antes de declará-la. Isso torna o código mais fácil de entender e menos propenso a erros. Além disso, as variáveis declaradas com `let` são acessíveis apenas no bloco de código onde foram declaradas, ou seja, se você declarar uma variável dentro de um bloco de código, ela não poderá ser acessada fora desse bloco. Você entenderá melhor o que isso quer dizer quando estudarmos estruturas de controle e funções.
3. **const**: A palavra-chave `const` também foi introduzida no JavaScript a partir da versão ES6 e é utilizada para declarar constantes. Se funcionamento é parecido com o da palavra chave `let`, porém, uma vez que uma constante é declarada, seu valor não pode ser alterado. Isso significa que você não pode atribuir um novo valor a uma constante depois de declará-la.

Veja abaixo a sintaxe para declarar variáveis com cada uma das palavras-chave:

```
var nome = 'João';
let idade = 30;
const PI = 3.1415;
```

Diferente de `let`e `var`, a palavra-chave `const` exige que a variável seja inicializada no momento da declaração, ou seja, você não pode declarar uma constante sem atribuir um valor a ela, além disso não é possível alterar seu valor, por isso os códigos abaixo resultariam em erros:

```
const PI;
PI = 3.1415; // Erro: SyntaxError: Missing initializer in const declaration
```

```
const X = 10;
X = 20; // Erro: TypeError: Assignment to constant variable.
```

Em JavaScript podemos ainda declarar diversas variáveis em uma única linha, separando-as por vírgula:

```
let nome = 'João', idade = 30, cidade = 'São Paulo';
```

Como JavaScript é uma linguagem de tipagem dinâmica, não precisamos informar o tipo de dado que a variável irá armazenar, o próprio JavaScript irá inferir o tipo de dado com base no valor que atribuímos à variável. Além disso, é possível alterar o tipo de dado armazenado em uma variável a qualquer momento, por exemplo:

```
let nome = 'João';
nome = 30; // Agora a variável nome armazena um número
```

Apesar de ser possível, alterar o tipo de dado de uma variável pode tornar o seu código confuso e mais propenso a erros, por isso é recomendado que você mantenha o tipo de dado armazenado em uma variável sempre o mesmo.

- Mais Informações sobre Variáveis

Operadores

Operadores são símbolos que representam diferentes tipos de operações que podem ser realizadas em JavaScript, como operações matemáticas, lógicas e de comparação. Os operadores são classificados de acordo com o número de operandos que eles recebem:

1. **Operadores Unários**: São operadores que recebem apenas um operando. Um exemplo de operador unário é o operador de negação `!`, que inverte o valor de uma expressão lógica (ex. `!true` é igual a `false`, e `!false` é igual a `true`).
2. **Operadores Binários**: São operadores que recebem dois operandos. Um exemplo de operador binário é o operador de adição `+`, que soma dois valores (ex. `2 + 3` é igual a `5`).
3. **Operadores Ternários**: São operadores que recebem três operandos. O único operador ternário em JavaScript é o operador condicional `? :`, que é utilizado para criar expressões condicionais (ex. `condição ? valor1 : valor2`). Nós exploraremos o operador condicional com mais detalhes quando estudarmos estruturas de controle.

A seguir, apresentamos os principais operadores da linguagem JavaScript, por categoria:

Operadores Aritméticos

Operador	Exemplo	Nome	Descrição
`+`	`2 + 3`	Adição	Realiza a adição entre dois valores
`-`	`5 - 3`	Subtração	Realiza a subtração entre dois valores
`*`	`2 * 3`	Multiplicação	Realiza a multiplicação entre dois valores
`/`	`6 / 2`	Divisão	Realiza a divisão entre dois valores
`%`	`5 % 2`	Resto da divisão	Retorna o resto da divisão entre dois valores
`++`	`x++`	Incremento	Incrementa o valor de uma variável em 1
`--`	`x--`	Decremento	Decrementa o valor de uma variável em 1

Operadores Lógicos

Operador	Exemplo	Nome	Descrição
`&&`	`true && false`	E	Retorna `true` se ambos os operandos forem verdadeiros
`||`	`true || false`	OU	Retorna `true` se pelo menos um dos operandos for verdadeiro
`!`	`!true`	Negação	Inverte o valor de uma expressão lógica

Operadores de Comparação

Operador	Exemplo	Nome	Descrição
`==`	`2 == 2`	Igualdade	Retorna `true` se os operandos forem iguais
`!=`	`2 != 3`	Diferença	Retorna `true` se os operandos forem diferentes
`===`	`2 === '2'`	Igualdade Estrita	Retorna `true` se os operandos forem iguais e do mesmo tipo
`!==`	`2 !== '2'`	Diferença Estrita	Retorna `true` se os operandos forem diferentes ou de tipos diferentes
`>`	`5 > 3`	Maior que	Retorna `true` se o operando da esquerda for maior que o operando da direita
`<`	`3 < 5`	Menor que	Retorna `true` se o operando da esquerda for menor que o operando da direita
`>=`	`5 >= 5`	Maior ou igual	Retorna `true` se o operando da esquerda for maior ou igual ao operando da direita
`<=`	`3 <= 5`	Menor ou igual	Retorna `true` se o operando da esquerda for menor ou igual ao operando da direita

Igualdade vs Igualdade Estrita: Em JavaScript, temos dois operadores de igualdade (e de diferença), o operador `==` e o operador `===`. O operador `==` realiza uma comparação de igualdade entre dois valores, porém, ele não leva em consideração o tipo de dado dos valores, ou seja, ele converte os valores para o mesmo tipo antes de realizar a comparação. Já o operador `===` realiza uma comparação de igualdade estrita, ou seja, ele compara os valores e os tipos dos valores, retornando `true` apenas se os valores forem iguais e do mesmo tipo.

```
2 == '2'; // true
2 === '2'; // false
```

```
2 != '2'; // false
2 !== '2'; // true
```

Para evitar erros de comparação, é recomendável utilizar os operadores de igualdade e diferença estrita (`===` e `!==`) sempre que possível.

Operadores de Atribuição

Operador	Exemplo	Nome	Descrição
`=`	`x = 10`	Atribuição	Atribui um valor a uma variável
`+=`	`x += 5`	Atribuição com Adição	Adiciona um valor a uma variável
`-=`	`x -= 5`	Atribuição com Subtração	Subtrai um valor de uma variável
`*=`	`x *= 5`	Atribuição com Multiplicação	Multiplica uma variável por um valor
`/=`	`x /= 5`	Atribuição com Divisão	Divide uma variável por um valor
`%=`	`x %= 5`	Atribuição com Resto da Divisão	Atribui o resto da divisão de uma variável por um valor

Operador de Concatenação

O operador `+`, além de ser utilizado para realizar operações matemáticas de adição, também é utilizado para concatenar strings em JavaScript. Quando utilizamos o operador `+` para concatenar strings, ele irá unir os valores das strings em uma única string.

```
let nome = 'João';
let sobrenome = 'Silva';
let nomeCompleto = nome + ' ' + sobrenome; // 'João Silva'
```

Assim, sempre que um dos operandos do operador `+` for uma string, o JavaScript irá realizar a concatenação dos valores ao invés da soma, isso quer dizer que no exemplo abaixo o valor da variável `resultado` será a string `'25'` e não o número `7`.

```
let numero = 2;
let resultado = numero + '5'; // '25'
```

Outros Operadores e Precedência de Operadores

A linguagem JavaScript possui diversos outros operadores que são utilizados para realizar diferentes tipos de operações. Além disso a linguagem possui regras de precedência que determinam a ordem em que os operadores são avaliados em uma expressão. Você pode encontrar uma lista completa de operadores e a ordem de precedência deles no site da Mozilla Developer Network.

Estruturas de Controle

As estruturas de controle são utilizadas para controlar o fluxo de execução de um programa, permitindo que você tome decisões com base em condições específicas. Quando escrevemos um algoritmo ele é executado de forma linear, uma linha após a outra, porém, em muitos casos precisamos que o programa tome decisões com base em determinadas condições. Para isso utilizamos as estruturas de controle que permitem que você determine quais partes do seu código devem ou não ser executadas.

Estrutura de Controle if (se)

A estrutura de controle `if` é utilizada para executar um bloco de código **se** uma determinada condição for verdadeira. A sintaxe da estrutura de controle `if` é a seguinte:

```
if (condição) {
    // Bloco de código a ser executado se a condição for verdadeira
}
```

A *condição* é uma expressão lógica que pode ser avaliada como verdadeira ou falsa. Se a condição for verdadeira, o bloco de código dentro das chaves `{}` será executado, caso contrário, o bloco de código será ignorado.

```
let idade = 18;

if (idade >= 18) {
    console.log('Você é maior de idade');
}
```

Nesse exemplo a variável idade armazena o valor 18, e a condição `idade >= 18` verifica se a idade é maior ou igual a 18. Como essa condição é verdadeira (`idade` é igual a 18), a nossa condição é verdadeira e o bloco de código dentro das chaves `{}` é executado, exibindo a mensagem `'Você é maior de idade'` no console.

Condições Verdadeiras e Falsas Em JavaScript, diferente de muitas outras linguagens, uma condição não precisa resultar no valor booleano `true` ou `false`, qualquer valor pode ser avaliado como verdadeiro ou falso. Valores que são avaliados como falsos são chamados de *falsy* e valores que são avaliados como verdadeiros são chamados de *truthy*.

Esse conceito é motivo de muitas confusões para quem está começando na linguagem JavaScript, por isso é importante entender quais valores são avaliados como verdadeiros e quais são avaliados como falsos, e sempre que possível utilizar condições que resultem em valores booleanos. Abaixo estão listados os valores que são avaliados como falsos em JavaScript:

- `false`
- `0`
- `''` (string vazia)
- `null`
- `undefined`
- `NaN` (Not a Number, valor resultante de operações matemáticas inválidas)

Condições Compostas Além das condições simples como vimos no exemplo anterior, é possível criar condições compostas utilizando os operadores lógicos `&&` (E) e `||` (OU). Ou seja, uma condição composta é um conjunto de condições que precisam ser satisfeitas para que o bloco de código seja executado. O resultado da combinação de duas condições é determinado pelo operador lógico utilizado. Veja o exemplo abaixo:

```
let idade = 18;
let possuiCNH = true;

if (idade >= 18 && possuiCNH) {
    console.log('Você pode dirigir');
}
```

Nesse exemplo, a condição `idade >= 18 && possuiCNH` verifica se a idade é maior ou igual a 18 **e** se a pessoa possui CNH. Se ambas as condições forem verdadeiras, o bloco de código dentro das chaves `{}` será executado, exibindo a mensagem `'Você pode dirigir'` no console. Perceba que não é necessário comparar a variável `possuiCNH` com `true`, pois o próprio valor da variável já é avaliado como verdadeiro.

```
let hora = 14;
let estaChovendo = true;
```

```
if (hora >= 18 || estaChovendo) {
  console.log('Ligue os faróis');
}
```

Nesse exemplo, a condição `hora >= 18 || estaChovendo` verifica se a hora é maior ou igual a 18 **ou** se está chovendo. Se qualquer uma das condições for verdadeira, o bloco de código dentro das chaves `{}` será executado, exibindo a mensagem `'Ligue os faróis'` no console.

Estrutura de Controle else (senão) A estrutura de controle `else` é utilizada para executar um bloco de código **se** a condição do `if` for falsa. Ou seja, a estrutura `else` sempre é utilizada em conjunto com a estrutura `if`, e é executada quando a condição do `if` é falsa. A sintaxe da estrutura de controle `else` é a seguinte:

```
if (condição) {
    // Bloco de código a ser executado se a condição for verdadeira
} else {
    // Bloco de código a ser executado se a condição for falsa
}
```

```
let idade = 17;

if (idade >= 18) {
    console.log('Você é maior de idade');
} else {
    console.log('Você é menor de idade');
}
```

Nesse exemplo, a variável idade armazena o valor 17, e a condição `idade >= 18` verifica se a idade é maior ou igual a 18. Como essa condição é falsa (`idade` é igual a 17), o bloco de código dentro das chaves `{}` do `else` é executado, exibindo a mensagem `'Você é menor de idade'` no console.

Estrutura de Controle else if (senão se) A estrutura de controle `else if` é utilizada para executar diferentes blocos de código com base em diferentes condições exclusivas. A estrutura `else if` é utilizada entre as estruturas `if` e `else`, e é executada se a condição do `if` for falsa e a condição do `else if` for verdadeira. A sintaxe da estrutura de controle `else if` é a seguinte:

```
if (condição1) {
    // Bloco de código a ser executado se a condição1 for verdadeira
} else if (condição2) {
    // Bloco de código a ser executado se a condição2 for verdadeira
} else {
    // Bloco de código a ser executado se nenhuma das condições anteriores for verdadeira
}
```

```
let idade = 21;

if (idade >= 18) {
    console.log('Você é maior de idade');
} else if (idade >= 16) {
    console.log('Você é adolescente');
} else {
    console.log('Você é criança');
}
```

Nesse exemplo, a variável idade armazena o valor 21, e a condição `idade >= 18` verifica se a idade é maior ou igual a 18. Como essa condição é verdadeira, o bloco de código dentro das chaves `{}` do `if` é executado,

exibindo a mensagem `'Você é maior de idade'` no console. Se a condição do `if` for falsa, a condição do `else if` é verificada, e se for verdadeira, o bloco de código dentro das chaves `{}` do `else if` é executado, exibindo a mensagem `'Você é adolescente'` no console. Se nenhuma das condições anteriores for verdadeira, o bloco de código dentro das chaves `{}` do `else` é executado, exibindo a mensagem `'Você é criança'` no console.

Pergunta: Se você trocar a ordem das condições `idade >= 16` e `idade >= 18`, o que aconteceria?

Estruturas de Controle Aninhadas As estruturas de controle `if`, `else if` e `else` podem ser aninhadas, ou seja, podemos utilizar uma estrutura de controle dentro de outra. Isso é útil quando precisamos verificar múltiplas condições de forma mais complexa. Veja o exemplo abaixo:

```
let idade = 21;
let possuiCNH = true;

if (idade >= 18) {
    if (possuiCNH) {
        console.log('Você pode dirigir');
    } else {
        console.log('Você não pode dirigir');
    }
} else {
    console.log('Você é menor de idade');
}
```

Nesse exemplo, a variável idade armazena o valor 21 e a variável possuiCNH armazena o valor `true`. A primeira condição `idade >= 18` verifica se a idade é maior ou igual a 18, se essa condição for verdadeira, a estrutura de controle `if` aninhada é executada. Dentro dessa estrutura de controle aninhada, a condição `possuiCNH` verifica se a pessoa possui CNH, se essa condição for verdadeira, o bloco de código dentro das chaves `{}` é executado, exibindo a mensagem `'Você pode dirigir'` no console. Se a condição `possuiCNH` for falsa, o bloco de código dentro das chaves `{}` do `else` é executado, exibindo a mensagem `'Você não pode dirigir'` no console. Se a condição `idade >= 18` for falsa, o bloco de código dentro das chaves `{}` do `else` é executado, exibindo a mensagem `'Você é menor de idade'` no console.

Funções

Imagine que você está desenvolvendo um programa que precisa calcular a média das 4 notas bimestrais de um aluno. Você provavelmente desenvolveria algo parecido com o código abaixo:

```
let nota1 = 7;
let nota2 = 8;
let nota3 = 6;
let nota4 = 9;

let media = (nota1 + nota2 + nota3 + nota4) / 4;
```

Agora imagine que você precisa repetir esse mesmo processo para os 40 alunos de uma turma, ou quem sabe para centenas de alunos de uma escola. Você teria que repetir esse mesmo código várias vezes, o que rapidamente deixaria de ser prático, seu código ficaria muito extenso e difícil de manter. Quando um problema (ou uma parte do problema) se apresenta de forma repetitiva, em geral a melhor solução é criar uma função que resolva esse problema, e então poderemos utilizar essa função sempre que precisarmos resolver esse problema.

Uma função é basicamente um bloco de código identificado por um nome, o qual pode ter suas próprias entradas e saídas. Assim como podemos utilizar o valor de uma variável em diferentes partes do código apenas referenciando-a pelo nome, também podemos chamar uma função em diferentes partes do código utilizando seu nome. A sintaxe para declarar uma função é a seguinte:

```
function nomeDaFuncao(parametro1, parametro2, ...) {
    // Bloco de código a ser executado
    return valorDeRetorno;
}
```

- `nomeDaFuncao`: É o nome da função, que deve ser único e significativo, ou seja, deve representar o que a função faz. Geralmente utilizamos verbos para nomear funções, como `calcularMedia`, `exibirMensagem`, `validarUsuario`, etc.
- `parametro1, parametro2, ...`: São os parâmetros da função, que são valores que a função espera receber para realizar suas operações, ou seja, são as entradas da função. Os parâmetros são utilizados dentro da função da mesma forma que as variáveis. Uma função pode não ter nenhum parâmetro, ou pode ter quantos parâmetros forem necessários.
- `return valorDeRetorno`: A instrução `return` é utilizada para retornar um valor da função para o local onde a função foi chamada. O `valorDeRetorno` é o valor que a função irá retornar, e pode ser qualquer tipo de dado, como um número, uma string, um objeto, um array, etc. Em JavaScript uma função pode pode não retornar nenhum valor, ou retornar apenas um valor.

Vamos reescrever o código que calcula a média das notas dos alunos em uma função:

```
function calcularMedia(nota1, nota2, nota3, nota4) {
    let media = (nota1 + nota2 + nota3 + nota4) / 4;
    return media;
}
```

Nesse exemplo estamos declarando uma função chamada `calcularMedia` que recebe 4 parâmetros, que são as notas dos alunos. Dentro da função estamos calculando a média das notas e retornando esse valor. Para utilizar essa função, basta chamá-la passando os valores das notas como argumentos:

```
let mediaAluno1 = calcularMedia(7, 8, 6, 9);
console.log(mediaAluno1); // 7.5
let mediaAluno2 = calcularMedia(5, 6, 7, 8);
console.log(mediaAluno2); // 6.5
```

Nesse exemplo estamos chamando a função `calcularMedia` duas vezes, passando as notas dos alunos como argumentos. A função irá calcular a média das notas e retornar esse valor, que será armazenado nas variáveis `mediaAluno1` e `mediaAluno2`, e então exibimos esses valores no console. Quando a função é chamada, os parâmetros informados na chamada da função são atribuídos às variáveis dentro da função, essa atribuição é feita de acordo com a ordem dos parâmetros. Ou seja na primeira chamada da função `calcularMedia(7, 8, 6, 9)`, o valor `7` é atribuído à variável `nota1`, o valor `8` é atribuído à variável `nota2`, e assim por diante.

A linha `return media;` é responsável por retornar o valor da variável `media` para o local onde a função foi chamada. Assim, quando a função é chamada e o cálculo da média é feito, o valor da média é retornado e armazenado na variável `mediaAluno1` ou `mediaAluno2`, que então pode ser utilizado em outras partes do código.

Algumas funções não recebem parâmetros, e outras funções não retornam valores. Por exemplo, a função abaixo exibe uma mensagem no console, mas não recebe nenhum parâmetro e não retorna nenhum valor:

```
function exibirMensagem() {
    console.log('Olá, mundo!');
}
```

Se tentarmos receber um valor dessa função, atribuindo-o à variável mensagem, o valor da variável será `undefined`, pois a função não retorna nenhum valor:

```
let mensagem = exibirMensagem(); // Exibe 'Olá, mundo!' no console
console.log(mensagem); // Exibe 'undefined' no console
```

O mesmo ocorre com uma função que espera receber um parâmetro, mas em sua chamada não é passado nenhum valor para esse parâmetro. Nesse caso se tentarmos utilizar o parâmetro no corpo da função, o valor

será `undefined`:

```
function exibirMensagem(nome) {
    console.log('Olá, ' + nome + '!');
}

exibirMensagem(); // Exibe 'Olá, undefined!' no console
```

- Documentação sobre Funções

Escopo da Função

Quando declaramos uma função, os parâmetros dessa função e as variáveis declaradas no corpo da função somente são acessíveis dentro do corpo dessa função, ou seja, elas possuem escopo local. Escopo é o contexto onde uma variável ou função é acessível. Se tentarmos acessar uma variável declarada dentro de uma função fora do corpo da função, o JavaScript irá retornar um erro, pois essa variável não é acessível fora do escopo da função.

```
function exibirMensagem() {
    let mensagem = 'Olá, mundo!';
    console.log(mensagem);
}

exibirMensagem(); // Exibe 'Olá, mundo!' no console
console.log(mensagem); // Erro: ReferenceError: mensagem is not defined
```

Nesse exemplo, a variável `mensagem` é declarada dentro da função `exibirMensagem`, e por isso ela só é acessível dentro do corpo da função. Quando tentamos acessar a variável `mensagem` fora do corpo da função, o JavaScript retorna um erro, pois essa variável não é acessível fora do escopo da função.

- Documentação sobre Escopo

Escopo Global

Por outro lado, variáveis declaradas fora do corpo das funções são acessíveis em qualquer parte do código, ou seja, elas possuem escopo global e podem ser utilizadas tanto dentro quanto fora das funções. Variáveis declaradas fora do corpo das funções são chamadas de variáveis globais.

```
let mensagem = 'Olá, mundo!';
function exibirMensagem() {
    console.log(mensagem);
}

exibirMensagem(); // Exibe 'Olá, mundo!' no console
console.log(mensagem); // Exibe 'Olá, mundo!' no console
```

Desafios

1. Crie uma função chamada `obterMaior` que recebe 2 números como parâmetros e retorna o maior deles. Utilize essa função para encontrar o maior número entre 4 e 7.
2. Crie uma função chamada `mostrarOrdemCrescente` que recebe 3 números como parâmetros e exibe esses números em ordem crescente. Utilize essa função para exibir os números 5, 3 e 7 em ordem crescente.
3. Crie uma função chamada `cumprimentar` que recebe um nome como parâmetro e exibe a mensagem `'Olá, [nome]!'`. Caso o nome não seja informado a função deve exibir a mensagem `'Você não me disse o seu nome!'`. Chame essa função passando o nome `'João'` como parâmetro e sem passar nenhum nome como parâmetro.

4. Crie uma função chamada `nomeCompleto` que recebe 2 parâmetros, `nome` e `sobrenome`, e retorna o nome completo. Utilize essa função oara obter o nome completo de `'José'` e `'Souza'`.
5. Crie uma função chamada `verificarEstoque` que recebe 3 parâmetros, `quantidade`, `baixo` e `crítico`, e retorna `'Estoque normal'` se a quantidade for maior que o valor de `baixo`, `'Estoque baixo'` se a quantidade for menor ou igual ao valor de `baixo` e maior que o valor de `crítico`, e `'Estoque crítico'` se a quantidade for menor ou igual ao valor de `crítico`. Utilize essa função para verificar o estoque de 100 unidades, com valor baixo de 50 unidades e valor crítico de 20 unidades.

Desafio Extra

Você sabia que alguns tipos na linguagem JavaScript possuem suas próprias funções (chamadas de métodos)? Por exemplo, o tipo de dados `string` possui métodos que permitem manipular ou obter informações sobre strings. Um desses métodos é o método `toUpperCase`, que retorna a string em que ele foi chamado convertida para letras maiúsculas. Por exemplo, o código abaixo exibe a string `'OLÁ, MUNDO!'` no console:

```
let mensagem = 'Olá, mundo!';
let mensagemMaiuscula = mensagem.toUpperCase();
console.log(mensagemMaiuscula); // 'OLÁ, MUNDO!'
```

Você pode encontrar uma lista completa de métodos disponíveis para o tipo `string` no site da Mozilla Developer Network na lista `Métodos de Instância`. Clicando sobre o nome do método você consegue ver uma descrição detalhada do método, exemplos de uso e a compatibilidade com os diferentes navegadores.

Nesse desafio você deve desenvolver uma função chamada `abreviarNome` que recebe 2 parâmetros, `nome` e `sobrenome`, e retorna as iniciais do nome e do sobrenome em letras maiúsculas. Exemplo:

```
let joseSouza = abreviarNome('josé', 'souza');
console.log(joseSouza); // Deve exibir 'J.S.'

let mariaSilva = abreviarNome('maria', 'silva');
console.log(mariaSilva); // Deve exibir 'M.S.'
```

Para concluir esse desafio você vai precisar utilizar o método `toUpperCase` que exemplificamos acima, além de algum outro método que lhe permita obter o primeiro caractere de cada `string`. Existem diversos métodos que podem lhe ajudar nessa tarefa, pesquise na documentação da Mozilla Developer Network para encontrar o método que achar mais adequado.