



JAVASCRIPT IMPRESSIONADOR

Curso Completo de JavaScript da Hashtag Programação



JAVASCRIPT IMPRESSIONADOR | HASHTAG PROGRAMAÇÃO

SUMÁRIO

MÓDULO 2

INTRODUÇÃO AO JAVASCRIPT

01	SOBRE O CURSO	011
02	UMA BREVE HISTÓRIA DO JAVASCRIPT	012
03	O QUE PODEMOS FAZER COM JS?	014
04	JAVASCRIPT NA PRÁTICA	019
05	AS CARACTERÍSTICAS DO JS	021
06	PREPARANDO O SEU AMBIENTE	023
07	INTRODUÇÃO AO TERMINAL WINDOWS	032
08	INTRODUÇÃO AO JAVASCRIPT - DICAS PARA O DESENVOLVIMENTO	038
09	O QUE É PROGRAMAÇÃO?	041
10	OS ALGORITMOS E A LÓGICA DE PROGRAMAÇÃO	044

MÓDULO 3

DADOS, VARIÁVEIS E OPERAÇÕES

MÓDULO 3

DADOS, VARIÁVEIS E OPERAÇÕES

04	OPERADORES ARITMÉTICOS EM JAVASCRIPT	070
05	PADRORIZANDO O CÓDIGO	078
06	TIPOS DE DADOS: STRINGS E NUMBERS	086
07	NÚMEROS BINÁRIOS E BITS	091
08	BOOLEANS: VARIÁVEIS E OPERAÇÕES	095
09	OPERADORES DE COMPARAÇÃO EM JAVASCRIPT	099
10	UNDEFINED E NULL	104
11	OBJETCT EM JAVASCRIPT	108
12	ARRAY EM JAVASCRIPT	113
13	NOMENCLATURA E MUTABILIDADE	117
14	POR QUE "TIPOS REFERÊNCIAS"	121
15	AS CONSEQUÊNCIAS DA MANIPULAÇÃO DE REFERÊNCIAS	123

01	PRIMEIRO PROGRAMA	046
02	VARIÁVEIS E CONSTANTES	056
03	DECLARANDO E DEFININDO VARIÁVEIS	063



Módulo 2

INTRODUÇÃO AO JAVASCRIPT

INTRODUÇÃO AO JAVASCRIPT

INTRODUÇÃO AO JAVASCRIPT

Bem-vindo ao Curso de Programação em JavaScript!

Este curso foi desenvolvido para aqueles que desejam mergulhar no emocionante mundo da programação e explorar os fundamentos essenciais desta linguagem de programação poderosa e versátil. Antes de adentrarmos nos detalhes específicos do JavaScript, é fundamental compreender o que é programação e como as linguagens de programação facilitam a comunicação entre humanos e máquinas.

Programação, em seu núcleo, é o processo de instruir um computador para realizar tarefas específicas de acordo com uma sequência lógica de comandos. É a arte de escrever algoritmos e códigos que orientam a máquina na execução de funções desejadas. Em outras palavras, é o ato de traduzir problemas do mundo real em uma linguagem que os computadores possam entender e resolver. Uma linguagem de programação é o meio através do qual os desenvolvedores comunicam suas instruções para o computador. Ela serve como um intermediário entre a mente humana e a máquina. Enquanto os humanos preferem se expressar em linguagens naturais, como inglês, espanhol ou português, as máquinas só podem entender instruções codificadas em uma linguagem específica, como JavaScript, Python, Java, entre outras.

O **JavaScript** é uma das linguagens de programação mais amplamente utilizadas no desenvolvimento web. Ela permite a criação de páginas web dinâmicas e interativas, adicionando comportamentos específicos aos elementos de uma página. Ao aprender JavaScript, você estará equipado para construir desde simples scripts até aplicações web complexas, abrindo um mundo de possibilidades no universo do desenvolvimento de software.

Neste curso, iremos explorar desde os conceitos básicos até as técnicas mais avançadas do JavaScript, capacitando você a escrever código eficiente e elegante, além de fornecer as habilidades necessárias para enfrentar desafios reais de programação. Prepare-se para embarcar em uma jornada de aprendizado emocionante e transformador. Vamos começar!

Nos primórdios da internet, a web era predominantemente estática, composta por páginas estáticas de texto e imagens. Porém, à medida que a demanda por interatividade e dinamismo crescia, os desenvolvedores procuravam maneiras de tornar as páginas web mais envolventes e funcionais. Foi nesse cenário que surgiu o JavaScript.

Criado por Brendan Eich, então engenheiro da Netscape, em 1995, o JavaScript foi concebido como uma linguagem de script leve e dinâmica para ser executada no navegador do usuário. Seu objetivo era permitir a interação do usuário com as páginas web, adicionando comportamentos e funcionalidades dinâmicas. Originalmente, o JavaScript foi chamado de LiveScript, mas foi renomeado para JavaScript por motivos de marketing, capitalizando a popularidade da linguagem Java na época.

Desde seu lançamento, o JavaScript passou por uma evolução notável. A introdução do ECMAScript, um padrão que define a especificação da linguagem JavaScript, ajudou a solidificar sua base e a estabelecer uma direção clara para o seu desenvolvimento.

No início dos anos 2000, um evento significativo marcou a história do JavaScript. Em 2008, o Google lançou o Chrome, seu próprio navegador, com um mecanismo de JavaScript chamado V8. O V8 trouxe uma melhoria significativa no desempenho do JavaScript, tornando-o muito mais rápido e eficiente. Esse avanço foi crucial para impulsionar o uso do JavaScript não apenas em páginas web, mas também em aplicativos web mais complexos.

Outro marco importante foi o lançamento do jQuery em 2006. O jQuery é uma biblioteca JavaScript que simplifica a manipulação do HTML, eventos, animações e interações AJAX. Com sua sintaxe simplificada e recursos poderosos, o jQuery facilitou o desenvolvimento web e ajudou a popularizar o JavaScript entre os desenvolvedores.

Além disso, em 2009, Ryan Dahl criou o Node.js, uma plataforma construída sobre o motor V8 do Chrome que permite executar JavaScript no lado do servidor. Isso revolucionou o desenvolvimento de aplicativos web, possibilitando o uso de JavaScript tanto no frontend quanto no backend, resultando em uma stack de desenvolvimento mais coesa e eficiente.

Hoje, o JavaScript é uma das linguagens de programação mais amplamente utilizadas em todo o mundo. Ele não só é essencial para o desenvolvimento web, mas também é amplamente empregado em outros domínios, como o desenvolvimento de aplicativos móveis, jogos, servidores e até mesmo em projetos de Internet das Coisas (IoT). Sua versatilidade, combinada com uma comunidade vibrante e recursos em constante expansão, tornam o JavaScript uma ferramenta indispensável para qualquer desenvolvedor moderno.

À medida que continuamos a avançar no cenário da tecnologia, o JavaScript permanece como um pilar fundamental, impulsionando a inovação e moldando a experiência online de bilhões de usuários em todo o mundo. Sua história é uma narrativa fascinante de perseverança, evolução e impacto duradouro na forma como interagimos com a internet.

Neste curso, você embarcará em uma jornada completa pelo universo do desenvolvimento web, começando do básico e avançando até tecnologias modernas e poderosas. Vamos explorar:

- **Desenvolvimento Web Básico:**

Entenda os fundamentos da web, incluindo HTML, CSS e as bases de como criar páginas estruturadas, estéticas e funcionais.

- **Fundamentos do JavaScript:**

Aprenda a programar com JavaScript, a linguagem que dá vida às páginas web, manipulando elementos, eventos e tornando os sites interativos.

- **Front-End Avançado:**

Aprofunde-se na construção de interfaces dinâmicas, componentes reutilizáveis e estilização avançada com bibliotecas e frameworks como React.

- **Back-End:**

Descubra como criar servidores, trabalhar com bancos de dados e conectar o front-end com o back-end, tornando suas aplicações completas e robustas.

- **React:**

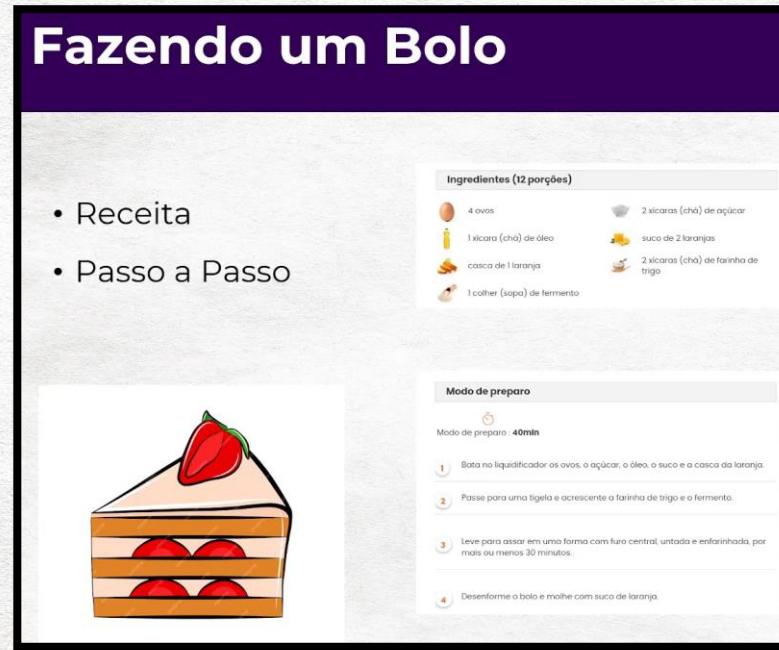
Domine o framework que revolucionou o desenvolvimento front-end, criando interfaces modernas, performáticas e interativas.

- **React Native:**

Leve suas habilidades para o próximo nível, desenvolvendo aplicativos móveis multiplataforma com a familiaridade do React. Ao final do curso, você terá as ferramentas e o conhecimento necessários para criar projetos profissionais tanto para web quanto para dispositivos móveis.

O Que É Programação?

Programação é o processo de criar um conjunto de instruções que um computador pode seguir para executar uma tarefa específica. Imagine que você está dando instruções para preparar um bolo de laranja delicioso - você precisa ser muito claro e específico para que o bolo saia perfeito. Da mesma forma, ao programar, você está basicamente ensinando ao computador o que fazer, passo a passo, assim como numa receita de bolo.



Como o Computador Executa Instruções: O Papel do Software e dos Programas

Quando você dá instruções para um computador, está essencialmente criando um conjunto de comandos que ele pode entender e seguir para realizar uma determinada tarefa. Para entender melhor esse processo, imagine que você está preparando uma receita de bolo de laranja. Cada passo na receita é como um comando para o computador, onde você precisa ser claro e específico para alcançar o resultado desejado.

No mundo da computação, esses comandos são chamados de programas ou software. Um programa é um conjunto de instruções escritas em uma linguagem específica que o computador pode entender e executar. Assim como uma receita de bolo, um programa é uma sequência de passos que dizem ao computador exatamente o que fazer, desde operações simples até tarefas mais complexas.

Quando você executa um programa em seu computador, o sistema operacional é responsável por coordenar e controlar a execução dessas instruções. Ele garante que o programa tenha acesso aos recursos necessários, como memória e processamento, e que as operações sejam executadas corretamente.

Cada programa que você usa em seu computador, seja um navegador da web, um editor de texto ou um jogo, é apenas uma série de instruções que dizem ao computador como realizar uma determinada tarefa. Assim como diferentes receitas produzem resultados diferentes na cozinha, diferentes programas produzem diferentes resultados no mundo digital.

Portanto, quando você está programando, está essencialmente escrevendo essas instruções para o computador seguir. É como ser o chef que cria uma nova receita - você precisa ser claro, lógico e preciso para garantir que o computador produza o resultado desejado. E assim como na cozinha, com prática e experiência, você se tornará mais habilidoso em criar programas eficientes e poderosos.



O que é um Algoritmo?

Um algoritmo é um conjunto de passos ou instruções que são seguidos para resolver um problema ou realizar uma tarefa. É como uma receita de bolo - você segue os passos na ordem correta para obter o resultado desejado.

Algoritmos em JavaScript:

JavaScript é uma linguagem de programação que permite escrever algoritmos para resolver problemas computacionais. Vamos começar com um exemplo simples de um algoritmo que soma dois números.

Exemplo de Algoritmo em JavaScript:

```
// Passo 1: Definir as variáveis
var numero1 = 5;
var numero2 = 3;
var soma;

// Passo 2: Realizar a operação
soma = numero1 + numero2;

// Passo 3: Exibir o resultado
console.log("A soma é: " + soma);
```

Neste exemplo, o algoritmo é dividido em três passos:

- 1. Definir as variáveis:** Aqui, estamos declarando duas variáveis **numero1** e **numero2**, e também uma variável **soma** para armazenar o resultado da operação.
- 2. Realizar a operação:** Adicionamos **numero1** e **numero2** e armazenamos o resultado em **soma**.
- 3. Exibir o resultado:** Usamos **console.log()** para mostrar o resultado da soma na tela.

A programação é a habilidade de dar instruções a um computador para que ele realize tarefas específicas. Por meio de linguagens como Javascript, podemos criar soluções que variam desde simples cálculos até sistemas complexos, como aplicativos web, jogos e automações.

Javascript é uma das linguagens mais versáteis e amplamente utilizadas, sendo essencial para o desenvolvimento web. Com ele, você pode criar páginas interativas, manipular dados, e até desenvolver backends robustos. Para iniciarmos sua jornada no mundo da programação, vamos explorar os pilares fundamentais que sustentam todas as linguagens de programação, incluindo o Javascript.

Os Pilares da Programação: Uma Abordagem Fundamental

A programação é construída sobre conceitos universais que sustentam qualquer linguagem ou tecnologia utilizada. Antes de mergulharmos no JavaScript, é essencial compreender os **quatro pilares fundamentais** da programação: **Linguagem, Algoritmos e Lógica de Programação, Dados, e Organização**. Eles formam a base para o desenvolvimento de qualquer sistema computacional.

1. Linguagem

As linguagens de programação são ferramentas que usamos para nos comunicar com os computadores. Cada linguagem tem suas particularidades, mas todas permitem:

- **Escrever instruções** que o computador pode entender e executar.
- Representar **conceitos humanos** de maneira estruturada e formal.

No caso do JavaScript:

- Ele é uma linguagem de alto nível, interpretada e voltada para o desenvolvimento web.
- É usada para criar interfaces interativas, manipular elementos de páginas web, e até desenvolver servidores.

2. Algoritmos e Lógica de Programação

Um **algoritmo** é uma sequência de passos que resolve um problema ou realiza uma tarefa. A **lógica de programação** é a habilidade de organizar esses passos de maneira eficiente e correta.

Características de um bom algoritmo:

- **Clareza:** Deve ser fácil de entender.
- **Eficiência:** Realiza a tarefa utilizando o menor número de recursos possíveis.
- **Corretude:** Sempre produz o resultado esperado.

3. Dados

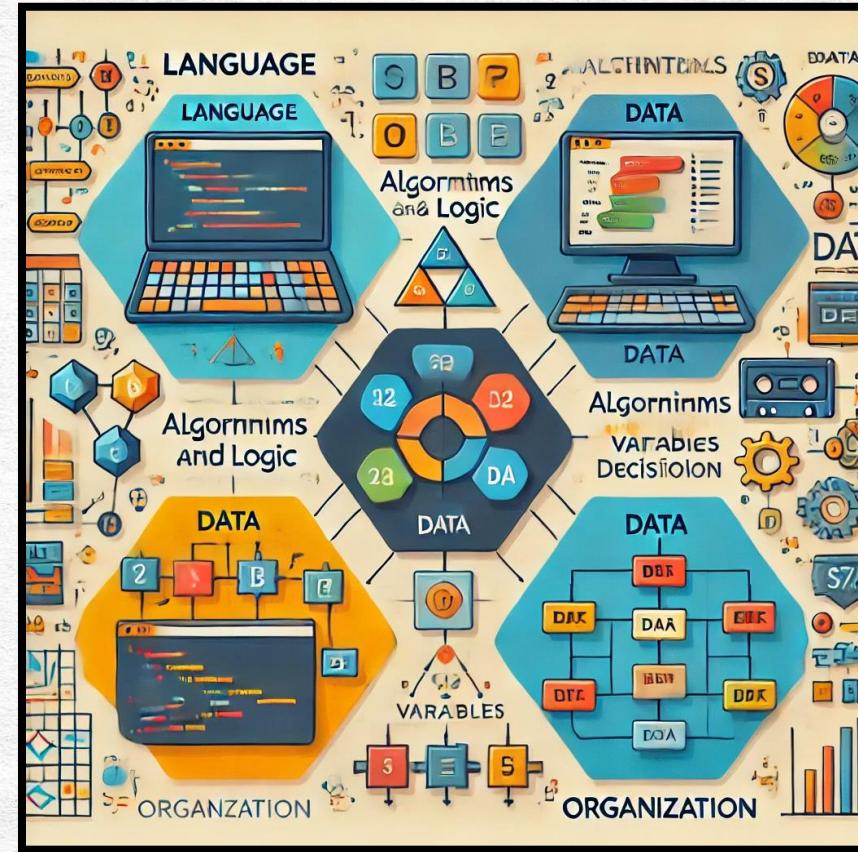
Os **dados** são o que alimentam os programas e representam informações do mundo real. Trabalhar com dados envolve:

- **Armazená-los:** Utilizando variáveis, arrays ou objetos.
- **Manipulá-los:** Executando operações como cálculos, filtros, e transformações.

4. Organização

- A organização de um código é crucial para que ele seja comprehensível e fácil de manter. Isso inclui:
- **Modularidade:** Dividir o programa em partes menores (funções, classes, módulos).
- **Boas práticas:** Usar nomes significativos, comentários, e padrões consistentes.
- A organização é essencial para o trabalho em equipe e a evolução de projetos ao longo do tempo.

Esses pilares – **linguagem, algoritmos e lógica de programação, dados e organização** – não só ajudam você a entender como a programação funciona, mas também tornam o aprendizado mais estruturado e eficiente. Ao longo deste curso, veremos como o JavaScript é aplicado em cada um desses aspectos, permitindo que você desenvolva tanto o raciocínio lógico quanto as habilidades práticas necessárias para criar aplicações incríveis!



A lógica de programação é a base para o desenvolvimento de sistemas, aplicações e soluções computacionais. Ela pode ser entendida como a habilidade de estruturar e resolver problemas de forma clara e eficiente, utilizando sequências lógicas e raciocínio.

O que é Lógica de Programação?

Lógica de programação é a aplicação do raciocínio para criar algoritmos que resolvem problemas ou realizam tarefas. Ela envolve identificar a melhor forma de executar uma ideia usando passos ordenados e precisos. Esses passos são codificados em uma linguagem de programação que o computador possa interpretar.

Por que é importante?

A lógica de programação é essencial porque:

- **Desenvolve o raciocínio:** Ajuda a pensar de maneira estruturada e a resolver problemas de forma mais eficiente.
- **Facilita o aprendizado de linguagens:** Com uma boa base em lógica, aprender novas linguagens de programação se torna mais simples.
- **Garante soluções eficientes:** Permite criar códigos mais claros, reutilizáveis e eficazes.

No estudo da lógica de programação com JavaScript, trabalhamos com diversos elementos fundamentais que formam a base para o desenvolvimento de aplicações. A seguir, exploraremos esses elementos em detalhes:

Variáveis

As variáveis são usadas para armazenar dados que podem ser manipulados e reutilizados ao longo do código.

Operadores

Os operadores permitem realizar operações matemáticas, comparações e manipulações lógicas.

Estruturas de Dados

JavaScript oferece estruturas para organizar e manipular conjuntos de dados.

Estruturas de Repetição

Essas estruturas permitem executar um bloco de código múltiplas vezes.

Estruturas Condicionais

As estruturas condicionais permitem executar blocos de código com base em condições

A Criação do Algoritmo

Um algoritmo é uma sequência de passos lógicos para resolver um problema. No contexto de JavaScript, criamos algoritmos para automatizar tarefas e manipular dados. É importante planejar antes de escrever o código:

- **Defina o problema:** O que precisa ser resolvido?
- **Planeje os passos:** Liste as etapas necessárias.
- **Implemente em código:** Converta os passos em JavaScript.
- **Teste e refine:** Execute o código e ajuste conforme necessário.

Compreender esses elementos é essencial para construir aplicações eficientes e resolver problemas de forma criativa.

Lógica de Programação

A lógica de programação é o processo de desenvolver algoritmos eficientes para resolver problemas por meio da aplicação de regras e procedimentos lógicos. Em outras palavras, é a capacidade de pensar de forma estruturada para encontrar soluções para problemas complexos, decompondo-os em etapas menores e mais gerenciáveis.

Relacionando isso com JavaScript, a lógica de programação em JavaScript envolve entender e aplicar os conceitos básicos da linguagem, como variáveis, estruturas de controle (como loops e condicionais), funções, arrays, objetos, entre outros. Isso significa que para desenvolver algoritmos em JavaScript, é necessário ter um entendimento sólido desses conceitos e saber como utilizá-los de forma eficaz para resolver problemas específicos.

A relação entre lógica de programação e algoritmos está na forma como os algoritmos são construídos utilizando-se da lógica de programação. Um algoritmo é uma sequência finita e ordenada de instruções que descrevem um processo computacional. A lógica de programação é a habilidade necessária para projetar e desenvolver esses algoritmos.

Identificar loja que mais faturou no período

- 1 - Ler os dados
- 2 - Agrupar vendas de cada loja (identificando a loja pelo nome) e somando o faturamento
- 3 - Dentre os 5 faturamentos totais do período, eu vou escolher o maior dentre eles. Farei isso ordenando os faturamentos do menor para o maior e então escolherei o último.

	A	B	C	D	E
1	Data	Loja	# de vendas	Faturamento Total	
2	12/10/2023	Rio Sul	184	1656,00	
3	12/10/2023	Botafogo Praia Shopping	224	1792,00	
4	12/10/2023	Shopping Leblon	208	2496,00	
5	12/10/2023	Shopping Nova América	192	1920,00	
6	12/10/2023	Shopping Tijuca	187	1683,00	
7	13/10/2023	Rio Sul	244	2196,00	
8	13/10/2023	Botafogo Praia Shopping	180	2160,00	
9	13/10/2023	Shopping Leblon	254	2540,00	
10	13/10/2023	Shopping Nova América	206	2266,00	

No estudo da lógica de programação com JavaScript, trabalhamos com diversos elementos fundamentais que formam a base para o desenvolvimento de aplicações. A seguir, exploraremos esses elementos em detalhes:

Um Problema como Desafio: Comissões de Vendas

Neste desafio, vamos lidar com o problema de calcular comissões para vendedores com base em suas vendas. A situação envolve uma tabela de vendas onde temos informações sobre:

- **Vendas realizadas.**
- **Vendedores.**
- **Comissões:** A serem calculadas com base nas metas atingidas.

Nosso objetivo é criar um algoritmo que resolva este problema de maneira clara e estruturada, aplicando os elementos da lógica de programação. Para isso, seguiremos os seguintes passos:

Passo a Passo do Algoritmo

- **Informar a lista de vendas:** Obter as vendas realizadas por cada vendedor.
- **Verificar a primeira venda:** Avaliar se a venda superou a meta estabelecida.
 - Caso sim, calcular a comissão como 15% do valor da venda.
 - Caso não, calcular a comissão como 10% do valor da venda.
- **Repetir o processo:** Continuar com o próximo item da lista até que todas as vendas sejam processadas.
- **Exibir o resultado:** Apresentar as comissões calculadas para cada vendedor.

Esse será o fluxo básico do nosso algoritmo.

Agora, vejamos como os elementos da lógica de programação se aplicam a cada etapa:

Elementos da Lógica de Programação no Algoritmo

- **Criação da Lista:** A lista é a estrutura de dados que armazenará as vendas de cada vendedor. Ela será utilizada para organizar e acessar as informações de maneira eficiente.
 - Por exemplo: [2000, 3000, 1500] poderia representar as vendas realizadas.
- **Condição para verificar metas:**
 - Se a venda for maior que a meta estabelecida, aplicamos 15%.
 - Caso contrário, aplicamos 10%.
- **Repetição para processar a lista:**
 - Usaremos uma estrutura de laço para iterar sobre cada item da lista até que todas as vendas sejam avaliadas.
- **Exibição dos Resultados:**
 - Depois de processar toda a lista, apresentamos as comissões de cada vendedor.

Vendedores	Vendas	Comissão
Daniel	R\$ 900,00	?
Alon	R\$ 1.000,00	?
Isadora	R\$ 1.100,00	?
Daniel M.	R\$ 850,00	?
Viviane	R\$ 950,00	?
Jorge	R\$ 900,00	?
Raphael	R\$ 700,00	?

Algoritmo

Passo a passo

1º passo: Informar a lista de vendas
2º passo: Verificar se a primeira venda superou a meta. Caso sim, calcular a comissão como 15% do valor.
3º passo: Caso não tenha batido a meta, a comissão do vendedor será calculada como 10% do valor vendido.
4º passo: Repetir a partir do passo 2 até terminar a minha lista.
5º Passo: Exibir o resultado

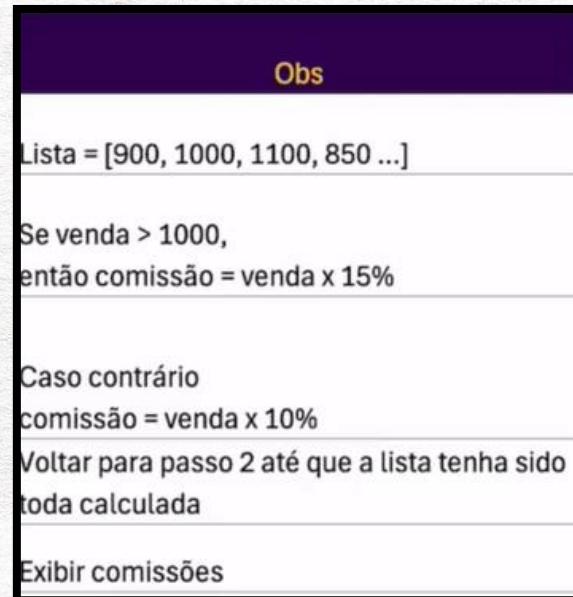


Planejamento do Algoritmo

Para resolver o problema com clareza:

- Primeiro, organizamos os dados em uma lista.
- Avaliamos cada item utilizando uma condição que determina a porcentagem da comissão.
- Iteramos sobre a lista para garantir que todas as vendas sejam processadas.
- Por fim, exibimos os resultados calculados.

Este planejamento nos ajuda a entender a aplicação prática dos conceitos de lógica de programação, preparando-nos para desenvolver o código em Javascript em etapas futuras.



Uma aplicação web é um software projetado para ser executado em um navegador da web e acessado pelos usuários através da internet. Em contraste com aplicativos tradicionais que são instalados localmente em um dispositivo, as aplicações web são hospedadas em servidores remotos e acessadas pelos usuários através de um navegador da web.

Essas aplicações são construídas usando tecnologias web padrão, como HTML, CSS e JavaScript, e são altamente interativas e dinâmicas. Elas podem variar desde sites simples, como blogs e lojas online, até aplicações web complexas, como plataformas de redes sociais, serviços de streaming de vídeo e sistemas de gerenciamento empresarial.

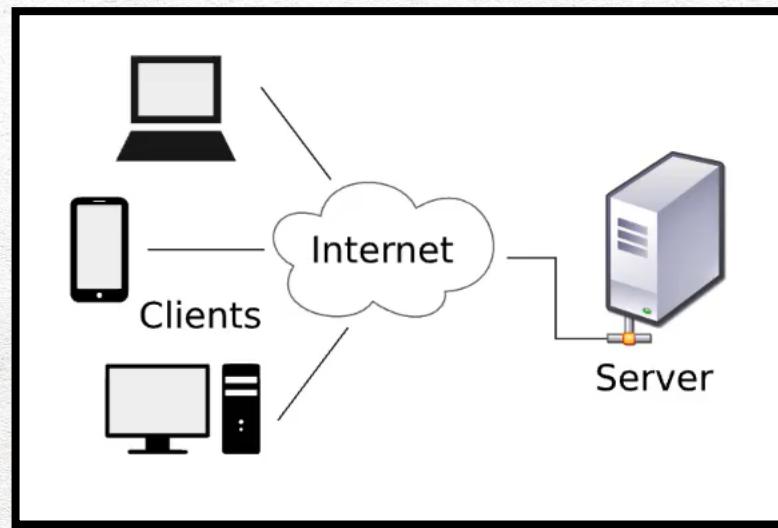
A comunicação entre o cliente (o navegador do usuário) e o servidor (onde a aplicação web está hospedada) é fundamental para o funcionamento de uma aplicação web. Esse processo geralmente ocorre através de um modelo de requisição e resposta, onde o cliente envia uma solicitação para o servidor e o servidor responde com os dados solicitados.

Quando um usuário interage com uma aplicação web, como clicar em um link ou preencher um formulário, o navegador do cliente envia uma requisição HTTP (Hypertext Transfer Protocol) para o servidor. Esta requisição contém informações sobre a ação do usuário e pode incluir dados adicionais, como parâmetros de pesquisa ou informações de formulário.

O servidor recebe a requisição do cliente e processa-a de acordo com a lógica da aplicação. Isso pode envolver a recuperação de dados de um banco de dados, a execução de cálculos ou o processamento de informações enviadas pelo cliente.

Após processar a requisição (**Request**), o servidor gera uma resposta (**Response**) que contém os dados solicitados ou informações sobre o resultado da ação realizada pelo usuário. Esta resposta é então enviada de volta ao cliente, que a exibe no navegador para o usuário final.

Essa troca contínua de requisições e respostas entre o cliente e o servidor permite que as aplicações web forneçam uma experiência interativa e dinâmica aos usuários, com atualizações em tempo real e funcionalidades avançadas que antes eram possíveis apenas em aplicativos tradicionais instalados localmente.



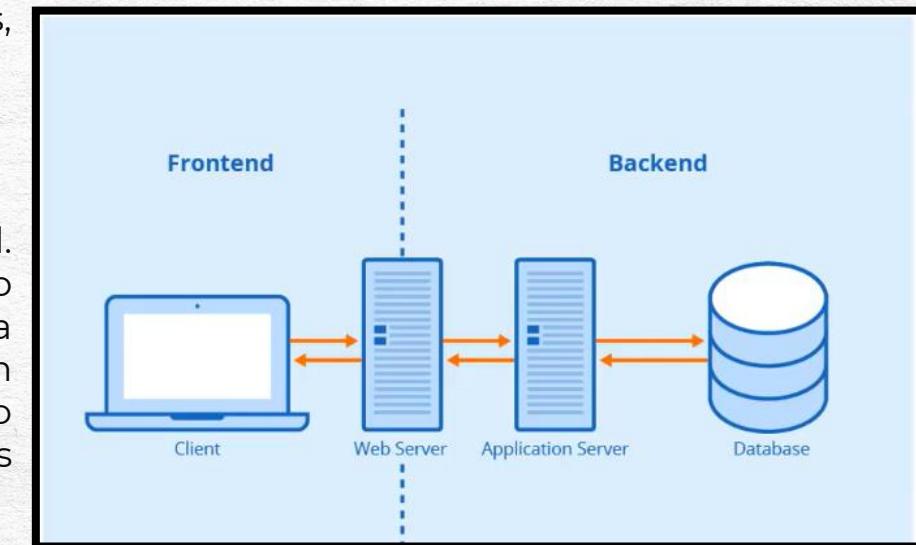
Aplicações Front-end

No front-end, JavaScript é empregado para manipular a interface do usuário (UI), interagir com HTML e CSS, controlar eventos do usuário, validar formulários, realizar requisições assíncronas ao servidor (AJAX), criar animações e efeitos visuais, além de armazenar dados localmente no navegador do cliente. Essas funcionalidades permitem criar interfaces dinâmicas, responsivas e altamente funcionais, proporcionando uma experiência de usuário interativa e atraente.

Aplicações Back-end

No desenvolvimento de aplicações web, o backend desempenha um papel crucial. JavaScript é uma linguagem poderosa também utilizada nessa área, permitindo o processamento de requisições, gerenciamento de dados, implementação de lógica de negócios e criação de APIs. Com tecnologias como Node.js, JavaScript oferece um ambiente robusto para construir aplicações escaláveis e eficientes no lado do servidor. Essas funcionalidades garantem o funcionamento adequado das aplicações web, proporcionando uma experiência de usuário fluida e responsiva.

Enquanto o front-end de uma aplicação web é responsável pela interface com o usuário e pela experiência visual, o backend é onde reside a inteligência e a lógica que impulsionam a aplicação. As aplicações no backend são essenciais para lidar com o processamento de dados, a lógica de negócios e a interação com bancos de dados e outros sistemas.



Desenvolvimento Web Full Stack

Desenvolvimento Full Stack refere-se à prática de construir aplicações web que abrangem tanto o lado do cliente (front-end) quanto o lado do servidor (back-end). Um desenvolvedor Full Stack é capaz de trabalhar em todas as camadas de uma aplicação web, desde a interface do usuário até o banco de dados e a lógica de negócios do servidor.

No contexto do desenvolvimento Full Stack, JavaScript desempenha um papel central. Ele é usado tanto no front-end quanto no back-end, graças ao Node.js, um ambiente de execução que permite executar JavaScript no servidor. Isso significa que um desenvolvedor Full Stack pode escrever código JavaScript para criar a interface do usuário interativa (UI) e também para implementar a lógica de negócios no servidor.

Os SmartPhones

JavaScript não é apenas uma linguagem de programação poderosa para computadores; ela também desempenha um papel crucial em dispositivos móveis, como smartphones. Quando você navega em páginas da web em seu smartphone ou usa aplicativos, é provável que esteja interagindo com JavaScript de várias maneiras.

No contexto de smartphones, JavaScript é usado para tornar as páginas da web responsivas e dinâmicas. Por exemplo, quando você rola para baixo em uma página da web em seu smartphone e elementos como menus, imagens e botões se ajustam automaticamente para se adequar ao tamanho da tela, isso é possível graças ao JavaScript. Ele permite que os desenvolvedores criem experiências de usuário adaptáveis e fluidas em dispositivos móveis.

Além disso, muitos aplicativos de celular são desenvolvidos usando tecnologias web, como HTML, CSS e JavaScript, em vez de linguagens de programação nativas específicas de plataforma. Esses aplicativos, conhecidos como aplicativos da web progressivos (PWAs), são executados em um navegador embutido no dispositivo e oferecem funcionalidades semelhantes às dos aplicativos nativos.

JavaScript é utilizado em aplicativos de celular para controlar a interação do usuário, realizar animações, realizar validações de formulários, fazer requisições assíncronas ao servidor, armazenar dados localmente e muito mais. Sua versatilidade e ubiquidade o tornam uma escolha popular para o desenvolvimento de aplicativos móveis modernos.

JOGOS

JavaScript também é amplamente utilizado no desenvolvimento de jogos, tanto para navegadores da web quanto para dispositivos móveis. Uma das principais vantagens do uso de JavaScript para jogos é sua acessibilidade. Como JavaScript é uma linguagem de programação amplamente utilizada na web, praticamente qualquer pessoa com conhecimento em programação web pode começar a desenvolver jogos imediatamente, sem a necessidade de aprender uma linguagem totalmente nova.

Além disso, o ecossistema JavaScript é rico em bibliotecas e frameworks dedicados ao desenvolvimento de jogos, como Phaser, Three.js, Babylon.js e Pixi.js. Essas bibliotecas fornecem conjuntos de ferramentas poderosos e fáceis de usar para criar gráficos, física, animações e interatividade em jogos.

JavaScript também é adequado para jogos simples e casuais que não requerem uma quantidade significativa de poder de processamento. Com a melhoria contínua do desempenho dos navegadores e o avanço da tecnologia web, JavaScript tornou-se capaz de lidar com jogos mais complexos e exigentes em termos de gráficos e desempenho.

Visual Studio Code

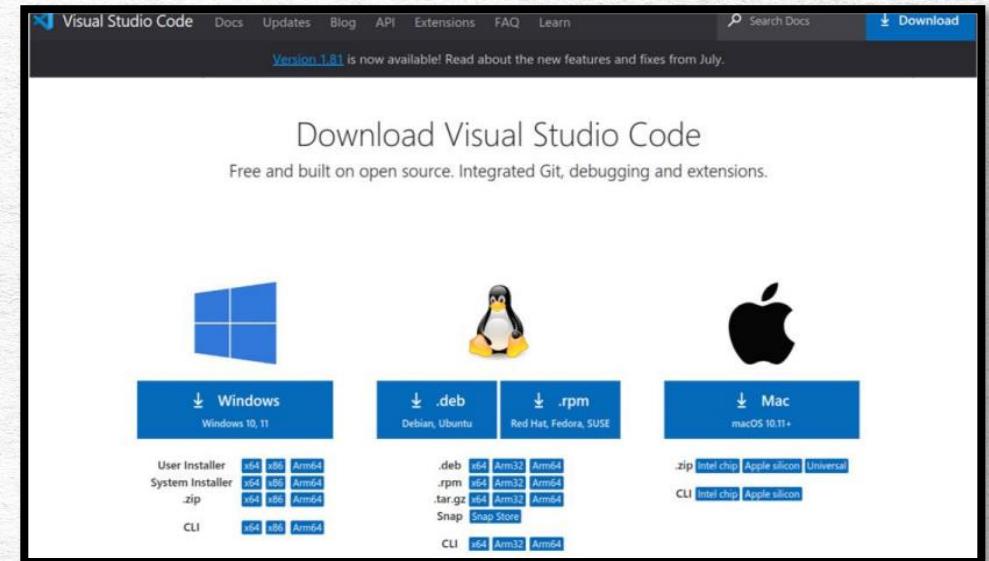
O Visual Studio Code, ou VS Code, é um editor de código que os programadores usam para escrever e editar seus programas. Ele é muito popular porque é fácil de usar e tem muitos recursos úteis. Um dos motivos para sua popularidade é o fato de que é muito bom para escrever JavaScript.

Com o VS Code, os programadores podem escrever, editar e testar seu código JavaScript de forma eficiente. O VS Code possui recursos que facilitam a escrita de código JavaScript, como sugestões automáticas de código, realce de sintaxe e depuração integrada, o que ajuda os programadores a identificar e corrigir erros em seu código JavaScript.

Caso você ainda não tenha o Visual Studio Code instalado , basta seguir os procedimentos abaixo. A instalação do VS Code é totalmente gratuita, e você pode usá-lo em seu computador sem precisar pagar nada. O link para fazer o download do programa é mostrado abaixo:

<https://code.visualstudio.com/>

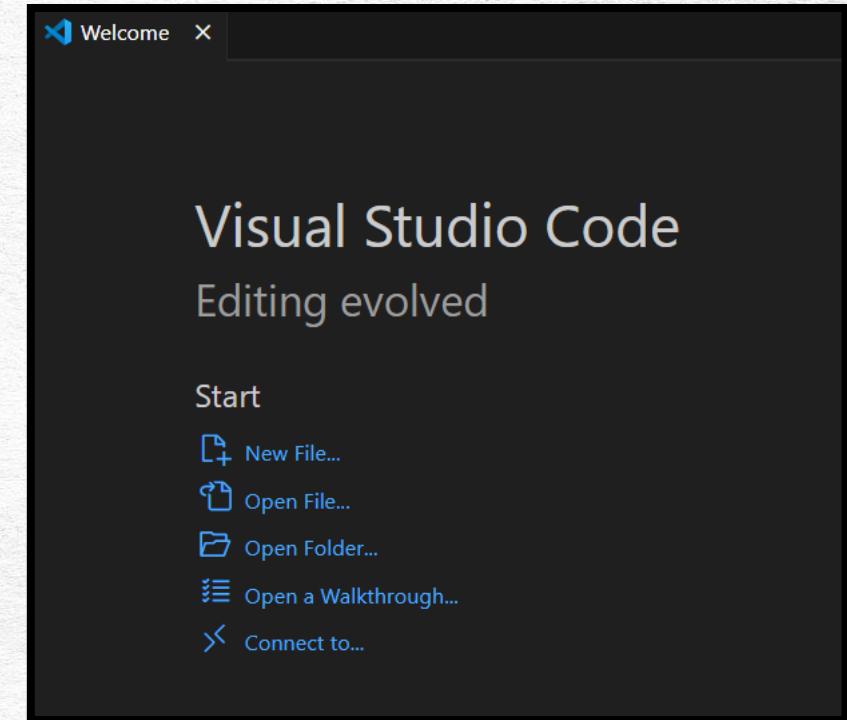
- 1 - Baixe o arquivo de instalação correspondente ao seu sistema operacional (Windows, MacOS ou Linux).
- 2 - Execute o arquivo de instalação e siga as instruções na tela. Em geral, é só continuar clicando em "Próximo".
- 3 - No Windows, durante a instalação, marque a opção "Add to path" para adicionar o VS Code às suas variáveis de ambiente.



Quando você abre o Visual Studio Code pela primeira vez, verá a sua janela principal com algumas opções importantes:

- **New File (Novo Arquivo):** Esta opção permite criar um novo arquivo em branco. É útil quando você deseja começar um novo projeto do zero e precisa de um arquivo para escrever seu código.
- **Open File (Abrir Arquivo):** Aqui, você pode selecionar um arquivo específico em seu computador para abrir no Visual Studio Code. É útil quando você já tem um arquivo existente e deseja editá-lo.
- **Open Folder (Abrir Pasta):** Esta opção permite abrir uma pasta inteira no Visual Studio Code. Quando você seleciona uma pasta, todos os arquivos contidos nela estarão disponíveis para edição no editor. É útil quando você está trabalhando em um projeto que possui vários arquivos e pastas relacionados.
- **Open a Walkthrough (Abrir um Tutorial):** O Visual Studio Code oferece tutoriais interativos, conhecidos como "walkthroughs", que ajudam os usuários a aprenderem a usar o editor e suas funcionalidades básicas. Esta opção permite acessar esses tutoriais para aprender mais sobre o funcionamento do VS Code.
- **Connect to (Conectar a):** Esta opção permite conectar o Visual Studio Code a diferentes serviços e recursos, como servidores remotos, contêineres Docker, ou mesmo a um ambiente de desenvolvimento remoto. Essa funcionalidade é útil para desenvolvedores que trabalham em ambientes distribuídos ou que precisam acessar recursos externos para desenvolver seus projetos.

Essas opções permitem que você comece a trabalhar no Visual Studio Code de várias maneiras, dependendo das suas necessidades específicas, seja começando um novo projeto, editando arquivos existentes, aprendendo a usar o editor ou conectando-se a recursos externos para desenvolvimento.



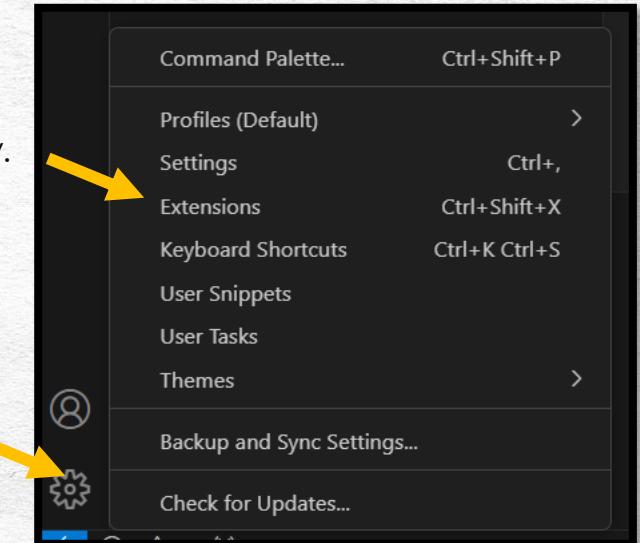
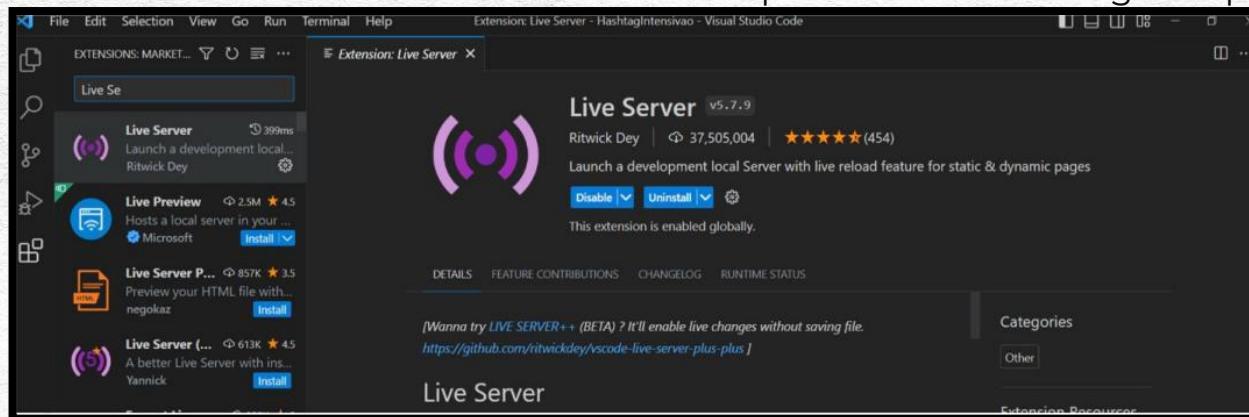
Extensões

As extensões no Visual Studio Code são pequenos programas que adicionam recursos extras ao editor de código. Elas são como "plugins" que podem ser instalados para personalizar e estender as funcionalidades do VS Code.

LIVE SERVER

O Live Server é uma extensão muito útil para desenvolvimento web, pois facilita a visualização e a atualização instantânea das alterações feitas no código HTML. Vou te explicar como baixar a extensão "Live Server" no VS Code:

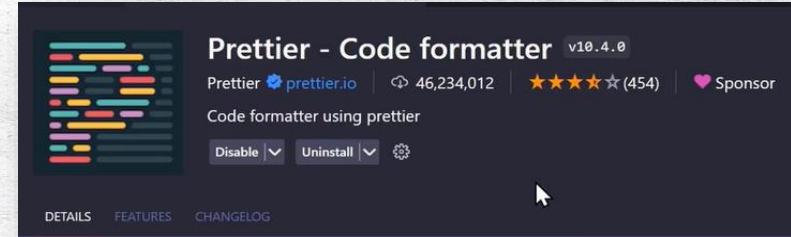
- Abra o Visual Studio Code (VS Code) e vá para a seção de extensões.
- Clique no ícone de extensões no menu lateral esquerdo (ou use o atalho "Ctrl+Shift+X").
- Pesquise por "Live Server" e clique em "Instalar" ao lado da extensão desenvolvida por Ritwick Dey.
- Aguarde a instalação e clique em "Gerenciar" para acessar as configurações da extensão.
- Personalize as configurações, se desejar.
- Abra um arquivo HTML no VS Code, clique com o botão direito do mouse e selecione "Abrir com Live Server".
- O Live Server iniciará um servidor local e abrirá o arquivo HTML no navegador padrão.



Prettier

O **Prettier** é uma extensão indispensável para desenvolvedores, pois automatiza a formatação do código, garantindo consistência e facilitando a leitura. Vou te explicar como instalar e configurar o **Prettier** no VS Code:

- **Acesse as Extensões no VS Code:**
 - Abra o Visual Studio Code (VS Code) e vá para a aba de extensões.
 - Clique no ícone de extensões no menu lateral esquerdo ou use o atalho Ctrl+Shift+X.
- **Instale a Extensão Prettier:**
 - Pesquise por "**Prettier - Code Formatter**" e clique em **Instalar** na extensão desenvolvida por **Prettier**.
- **Configure o Prettier como Formatador Padrão:**
 - Após a instalação, vá para as configurações do VS Code (Ctrl+, ou clique no ícone de engrenagem no canto inferior esquerdo).
 - Na barra de busca das configurações, digite "**Default Formatter**" e selecione o **Prettier** como seu formatador padrão.
- **Ative o Formato ao Salvar:**
 - Ainda nas configurações, procure por "**Format On Save**" e marque a opção para ativar a formatação automática sempre que salvar o arquivo.
- **Teste o Prettier no Código:**
 - Abra qualquer arquivo de código (HTML, CSS, JavaScript, etc.), bagunce um pouco a formatação e salve o arquivo (Ctrl+S). O Prettier irá automaticamente organizar o código para você!



Com o Prettier configurado, seu código ficará sempre limpo, organizado e padronizado, permitindo que você foque no que realmente importa: **o desenvolvimento!**

NODE.JS

O Node.js é um ambiente de execução de código JavaScript do lado do servidor. Ele permite que você execute código JavaScript fora do navegador, o que significa que você pode criar aplicativos de servidor, scripts de linha de comando e muito mais usando JavaScript.

Para instalar o Node.js, você pode seguir os seguintes passos:

- Acesse o site oficial do Node.js em <https://nodejs.org>.
- Na página inicial, você verá duas versões para download: LTS (Long Term Support) e Current. A versão LTS é recomendada para a maioria dos usuários, pois é mais estável e possui suporte a longo prazo. Selecione a versão LTS ou a versão mais recente, se preferir.
- Após selecionar a versão desejada, você será redirecionado para a página de download. Escolha o instalador adequado para o seu sistema operacional (Windows, macOS ou Linux) e clique no link para iniciar o download.
- Após o download ser concluído, execute o instalador e siga as instruções na tela para concluir a instalação.
- Após a instalação ser concluída, você pode verificar se o Node.js foi instalado corretamente abrindo o terminal ou prompt de comando e digitando o comando node -v. Se a versão do Node.js for exibida, significa que a instalação foi bem-sucedida.



No Windows, o termo "terminal" é frequentemente utilizado para se referir a interfaces de linha de comando (CLI, Command-Line Interface) como o Prompt de Comando (Command Prompt) ou o Windows PowerShell. Essas ferramentas permitem que os usuários interajam com o sistema operacional digitando comandos em vez de utilizar uma interface gráfica.

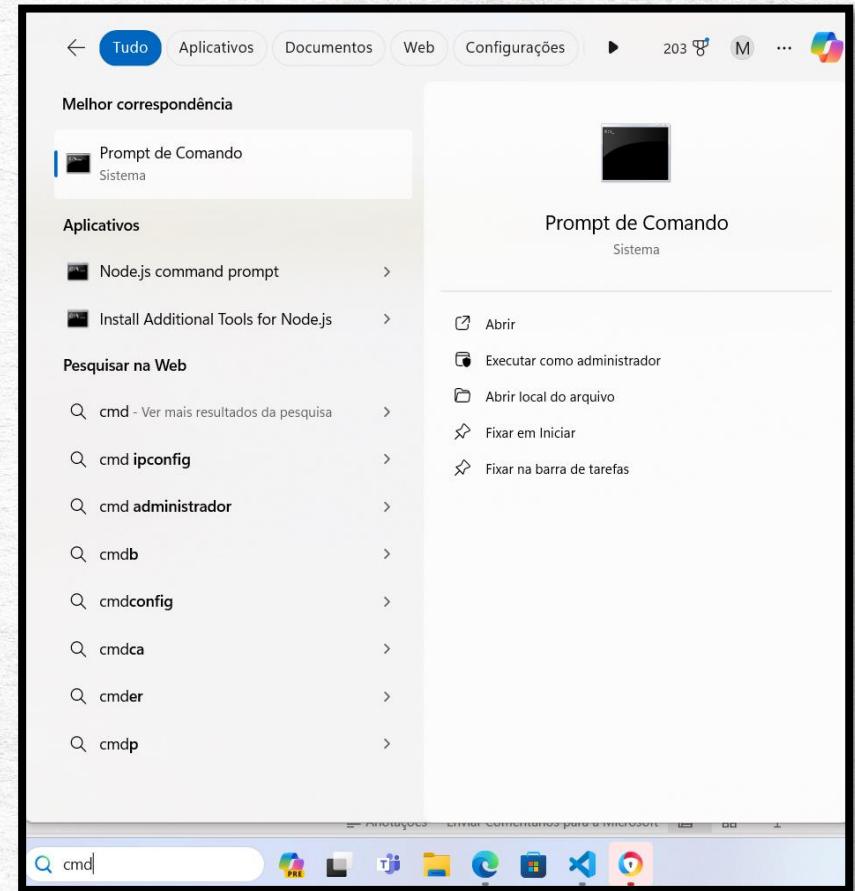
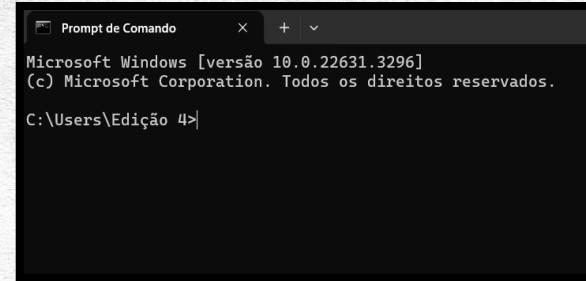
Aqui está uma breve explicação sobre cada um:

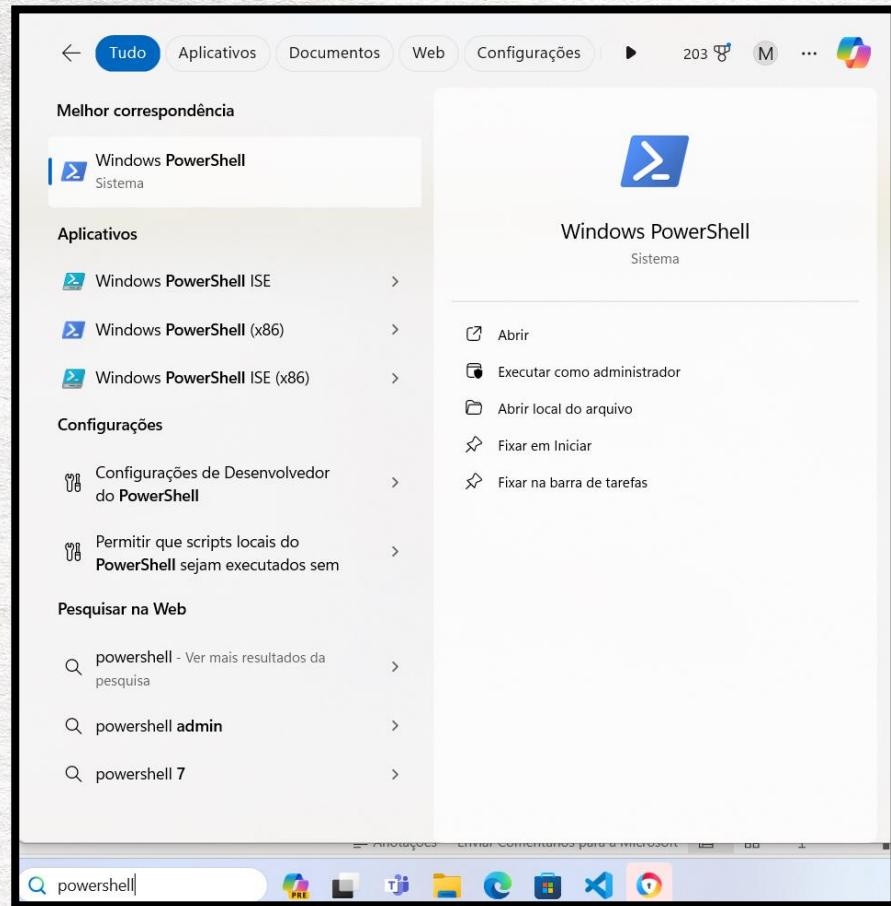
Prompt de Comando (cmd):

- O Prompt de Comando é uma interface de linha de comando padrão no Windows, existente desde as versões mais antigas do sistema operacional.
- Ele utiliza o interpretador de comandos cmd.exe e suporta uma variedade de comandos internos e externos.
- O Prompt de Comando geralmente possui uma interface de usuário mais básica em comparação com o PowerShell, mas ainda é amplamente utilizado para tarefas simples e compatibilidade com scripts mais antigos.

• Abrir o Prompt de Comando:

- Pressione **Win + R** para abrir a caixa de diálogo "Executar".
- Digite "cmd" e pressione Enter.
- Alternativamente, você pode pesquisar por "Prompt de Comando" no menu Iniciar.





• Windows PowerShell:

- O Windows PowerShell é uma poderosa interface de linha de comando desenvolvida pela Microsoft, introduzida inicialmente no Windows XP como uma opção de download, mas desde o Windows 7, tornou-se parte integrante do sistema operacional.
- Ele utiliza o interpretador de comandos powershell.exe e possui uma sintaxe mais avançada, bem como recursos mais robustos, como suporte a scripting, manipulação de objetos e acesso a APIs do sistema operacional.
- O PowerShell é amplamente adotado por administradores de sistemas e desenvolvedores devido à sua flexibilidade e capacidade de automação.

Para abrir o Prompt de Comando ou o PowerShell:

• Abrir o PowerShell:

- Pressione **Win + X** para abrir o menu de contexto do Windows.
- Selecione "Windows PowerShell" na lista de opções.
- Você também pode pesquisar por "PowerShell" no menu Iniciar.

Ambos os terminais oferecem funcionalidades distintas e são úteis para diferentes tipos de tarefas. A escolha entre eles depende das necessidades específicas do usuário e das tarefas que precisam ser executadas.

As **árvores de arquivos no Windows** são uma representação hierárquica dos diretórios (pastas) e arquivos armazenados no sistema de arquivos do Windows. Essa estrutura de árvore é organizada de forma que cada diretório pode conter outros diretórios e/ou arquivos. O topo da árvore é chamado de diretório raiz, que é representado por uma unidade de armazenamento, como C:, D:, etc. Aqui estão alguns conceitos importantes relacionados às árvores de arquivos no Windows:

Diretórios (Pastas):

- Os diretórios, ou pastas, são usados para organizar e agrupar arquivos relacionados.
- Cada diretório pode conter outros diretórios e/ou arquivos.
- O diretório raiz de cada unidade de armazenamento (geralmente representado por letras como C:, D:, etc.) é o ponto de partida da árvore de arquivos.

Arquivos:

- Os arquivos são unidades individuais de dados armazenados em dispositivos de armazenamento, como discos rígidos, unidades USB, etc.
- Eles podem ser de vários tipos, como documentos de texto, planilhas, imagens, executáveis de programas, entre outros.

Caminhos:

- Cada diretório e arquivo em uma árvore de arquivos do Windows tem um caminho único que indica sua localização na árvore.
- Um caminho é uma sequência de diretórios separados por barras invertidas (), onde o último elemento é o nome do arquivo ou diretório em questão.

Explorador de Arquivos:

- O Explorador de Arquivos é a interface gráfica padrão para navegar pela árvore de arquivos no Windows.
- Ele permite que os usuários visualizem, organizem, copiem, movam e excluam arquivos e diretórios usando uma interface de usuário amigável.
- Ao navegar pela árvore de arquivos no Windows, os usuários podem criar, renomear, copiar, mover e excluir diretórios e arquivos conforme necessário para organizar e gerenciar seus dados. A estrutura da árvore de arquivos facilita a localização e o acesso a diferentes arquivos e pastas no sistema operacional Windows.

Vamos relacionar as informações sobre árvores de arquivos do Windows com a utilização do terminal:

Navegação na Estrutura de Diretórios:

- Assim como no Explorador de Arquivos, no terminal você pode navegar pela estrutura de diretórios do Windows para acessar diferentes pastas e arquivos.
- Utilizando comandos como **cd** (change directory) no Prompt de Comando ou PowerShell, você pode se mover entre os diretórios da árvore de arquivos.

Caminhos de Arquivos e Diretórios:

- No terminal, você trabalha com caminhos de arquivos e diretórios para especificar a localização de pastas e arquivos.
- Por exemplo, **C:\Users\Username\Documents** é um caminho que aponta para a pasta "Documents" dentro do diretório do usuário.

Manipulação de Arquivos e Diretórios:

- Assim como no Explorador de Arquivos, você pode criar, renomear, copiar, mover e excluir arquivos e diretórios usando comandos no terminal.
- Comandos como **mkdir** (make directory), **rename**, **copy**, **move** e **del** (delete) permitem realizar essas operações no Prompt de Comando ou PowerShell.

Gerenciamento de Tarefas:

- O terminal no Windows também permite a execução de scripts e automação de tarefas, facilitando o gerenciamento de arquivos em grande escala.
- Você pode criar scripts em lote (.bat) para automatizar tarefas repetitivas no Prompt de Comando, ou scripts em PowerShell (.ps1) para tarefas mais avançadas e complexas no Windows PowerShell.

Em resumo, assim como no Explorador de Arquivos, no terminal do Windows você pode navegar pela estrutura de diretórios, trabalhar com caminhos de arquivos e diretórios, manipular arquivos e diretórios e automatizar tarefas de gerenciamento de arquivos. O terminal oferece uma interface de linha de comando poderosa para usuários que preferem trabalhar de forma mais eficiente e automatizada.

O **comando "cd" (change directory)** é uma ferramenta fundamental no terminal de sistemas operacionais baseados em Unix, como Linux e macOS, bem como no prompt de comando do Windows. Ele permite que os usuários naveguem pelo sistema de arquivos, mudando de diretório para outro. O comando "cd" é frequentemente utilizado para acessar diferentes pastas no sistema de arquivos.

Para utilizar o comando "cd", basta digitar "cd" seguido do nome do diretório para o qual você deseja navegar. Por exemplo, para entrar na pasta chamada "Documentos", você digitaria:

```
cd Documentos
```

Uma característica útil do comando "cd" é a capacidade de voltar um diretório usando "..". Por exemplo, se você estiver dentro da pasta "Documentos" e quiser voltar para a pasta anterior, poderá digitar:

```
cd ..
```

sso o levará de volta ao diretório pai da pasta atual. Você pode usar ".." repetidamente para subir vários níveis no sistema de arquivos.

Além disso, para voltar diretamente para o diretório inicial do usuário, você pode usar o comando "cd" sem argumentos:

```
cd
```

Este comando levará você de volta ao diretório inicial do usuário.

Outro comando útil relacionado à navegação de diretórios é "cd -", que o leva de volta para o último diretório em que você estava antes de mudar para o diretório atual. Isso pode ser particularmente útil se você estiver alternando entre dois diretórios com frequência.

O comando "dir" é utilizado nos sistemas Windows para listar os arquivos e diretórios em um determinado diretório no prompt de comando, fornecendo informações básicas como nome, tamanho e data de modificação.

```
Prompt de Comando
Microsoft Windows [versão 10.0.22631.3296]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Edição 4>dir
0 volume na unidade C não tem nome.
O Número de Série do Volume é E833-083B

Pasta de C:\Users\Edição 4

03/04/2024 22:34    <DIR>    .
03/04/2024 22:05    <DIR>    ..
18/12/2023 20:49    <DIR>    .vscode
29/11/2023 10:36    <DIR>    AppData
29/11/2023 10:37    <DIR>    Contacts
03/04/2024 22:22    <DIR>    Desktop
07/04/2024 20:30    <DIR>    Documents
08/04/2024 14:25    <DIR>    Downloads
29/11/2023 10:37    <DIR>    Favorites
29/11/2023 10:37    <DIR>    Links
29/11/2023 10:37    <DIR>    Music
04/04/2024 16:20    <DIR>    OneDrive
26/01/2024 20:11    <DIR>    Pictures
29/11/2023 10:37    <DIR>    Saved Games
29/11/2023 11:12    <DIR>    Searches
08/02/2024 23:53    <DIR>    Videos
              0 arquivo(s)          0 bytes disponíveis
              16 pasta(s)  431.241.981.952 bytes disponíveis
```

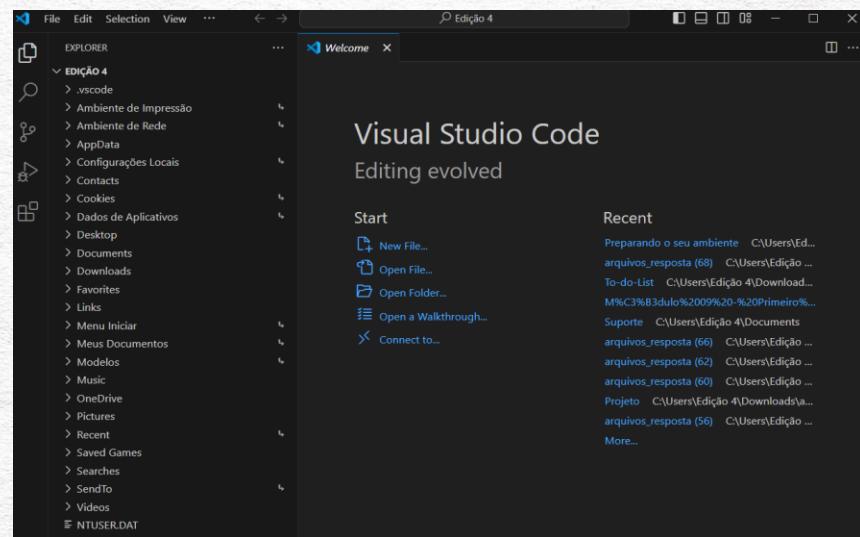
O comando "mkdir" (make directory) é utilizado no terminal para criar um novo diretório (pasta) no sistema de arquivos. Basta digitar "mkdir" seguido pelo nome do diretório que você deseja criar. Por exemplo, para criar um diretório chamado "novo_diretorio", você digitaria:

```
mkdir novo_diretorio
```

O comando "code ." é usado no terminal para abrir o Visual Studio Code no diretório atual. Isso abre o Visual Studio Code com o diretório atual como o projeto raiz, permitindo que você comece a trabalhar em arquivos desse diretório diretamente no editor.

```
Prompt de Comando
Microsoft Windows [versão 10.0.22631.3296]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Edição 4>code .
```



Módulo 3

DADOS, VARIÁVEIS E OPERAÇÕES

DADOS, VARIÁVEIS E OPERAÇÕES

DADOS, VARIÁVEIS E OPERAÇÕES

Neste módulo, mergulharemos nos conceitos essenciais para começar sua jornada na programação com Javascript. Vamos aprender sobre:

- **O que são dados em JavaScript.**
- **Como utilizar variáveis para armazenar informações.**
- **Criar e executar seu primeiro programa em JavaScript.**

Nosso objetivo é entender como representar e manipular informações no código, criando as bases para projetos mais complexos no futuro.

O que são dados em JavaScript?

Dados são as informações que usamos e manipulamos em nossos programas. Em JavaScript, essas informações podem assumir diferentes tipos, como:

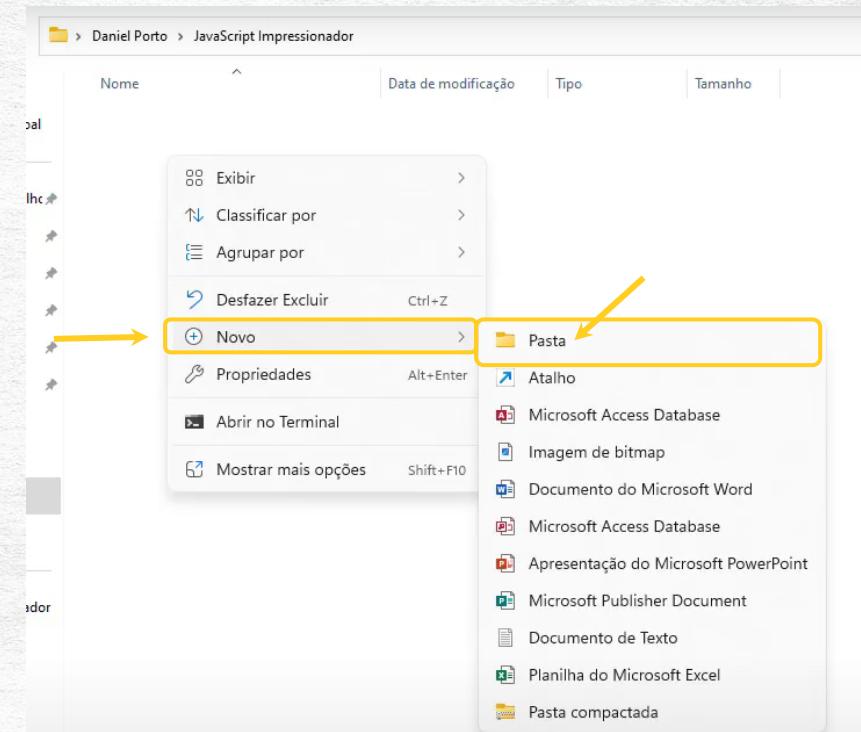
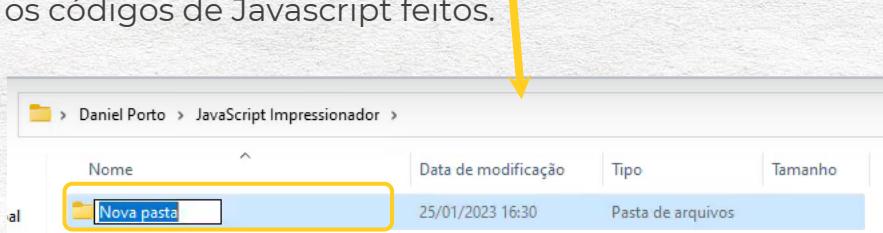
- **Números:** Representam valores numéricos.
 - Exemplo: 10, 3.14.
- **Strings:** Sequências de caracteres usadas para representar texto.
 - Exemplo: 'Olá, mundo! '.
- **Booleanos:** Representam valores verdadeiros ou falsos.
 - Exemplo: true, false.
- **Arrays e Objetos:** Estruturas que organizam dados de maneira mais complexa.

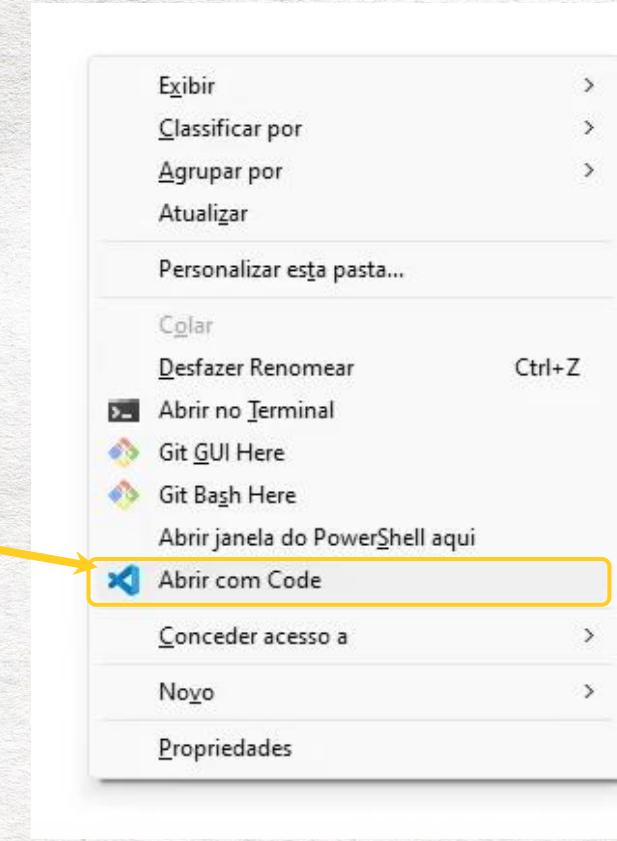
Esses dados são fundamentais para construir programas que interajam com usuários e realizem tarefas úteis.

Neste módulo, iremos escrever nossos primeiros códigos em JAVASCRIPT.

Para iniciarmos, vamos realizar os próximos passos para que os códigos e as aulas fiquem organizados em uma estrutura de pastas e de fácil compreensão.

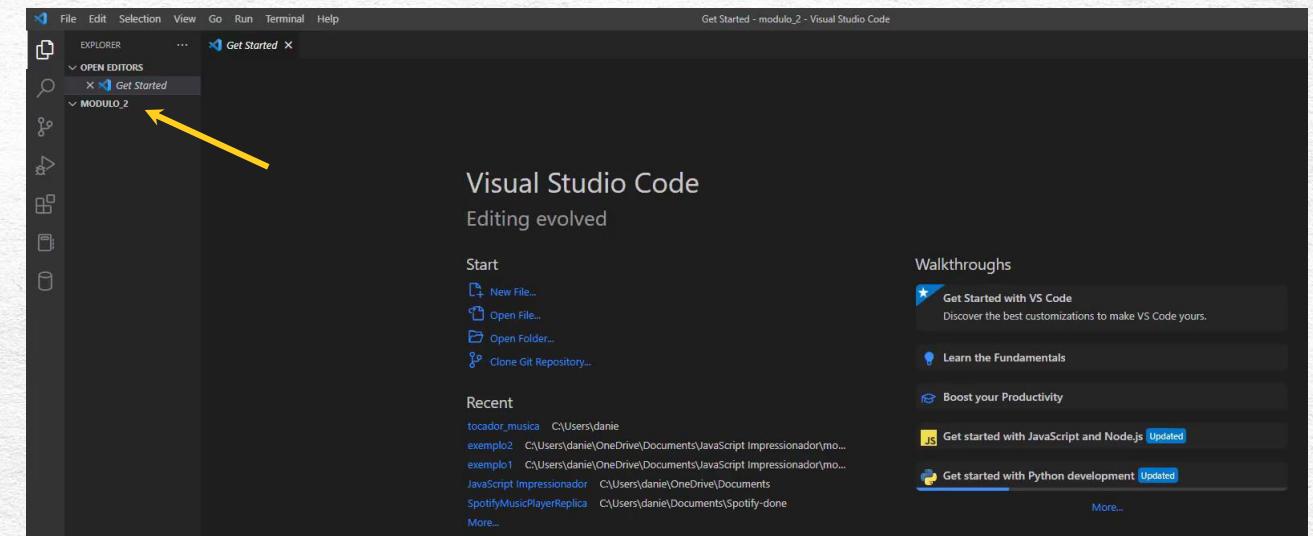
- **1º Passo:** Criaremos uma pasta chamada JavaScript Impressor no seu computador (no local que faça sentido para você- exemplo: Pasta Pessoal, Área de Trabalho, Documentos).
- **2º Passo:** Dentro da pasta JavaScript Impressor, clicando com o botão direito iremos clicar em ‘Novo’, depois ‘Pasta’, iremos nomear ela de acordo de onde você estará no curso, que nesse momento será ‘modulo_2’. Onde iremos armazenar todos os códigos de Javascript feitos.



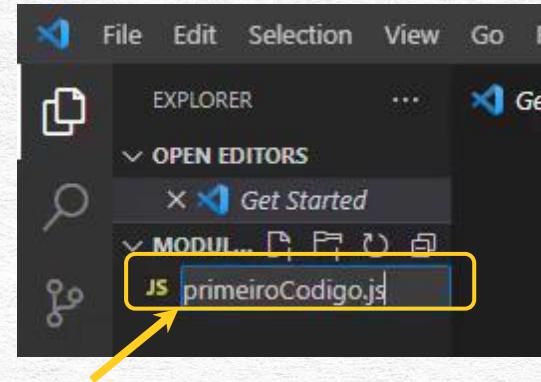
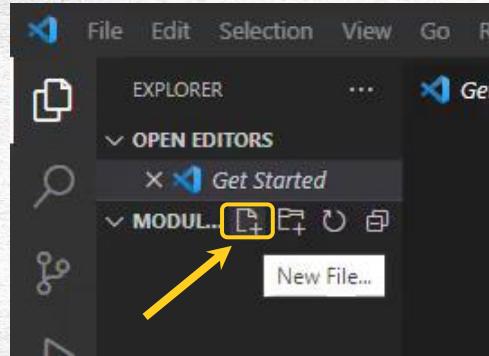


- **3º Passo:** Dentro da pasta modulo_2, iremos clicar com 'SHIFT + botão direito do mouse' e clicar na opção '**Abrir com Code**'.

Ao finalizar o 3º passo, o programa *Visual Studio Code (VS Code)* irá abrir “olhando” para esta pasta, ou seja, todo o código que trabalharmos no programa estará dentro da pasta que criamos. E assim a nossa organização e leitura ficará mais simples.



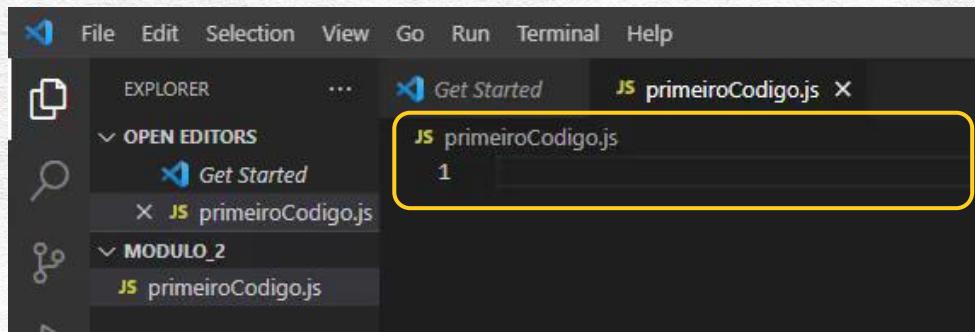
- 4º Passo: Iremos iniciar um novo arquivo, clicando no ícone de ‘New File’, nomeando-o “primeiroCodigo.js”



Repare que ao colocarmos “.js” no final do nosso arquivo e salvar um ícone JS foi gerado. Mas o que é e por que isso ocorre??

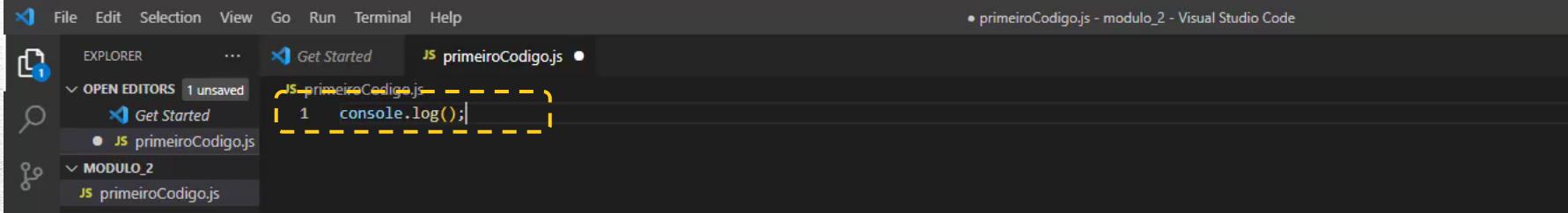
O VS Code, ao receber um arquivo “.js” automaticamente já faz a leitura de que se trata de um arquivo de código de Javascript, e isso ocorre porque essa extensão “.js” é utilizada para além de sinalizar, criar um arquivo do tipo de Javascript.

Isso facilita o VS Code a interpretar e tratar ele, gerando dicas e atalhos que facilitam o processo de escrever códigos.



Finalmente vamos dar início. O JavaScript é uma sequência de comandos a serem executados, ou seja, um comando JavaScript tem o propósito de dizer ao programa o que fazer, como se fosse uma instrução.

O comando **console.log()** imprime o texto no console como uma mensagem de log dentro da tela, basicamente as informações que colocamos dentro dos parentes serão impressas na saída do VS Code.



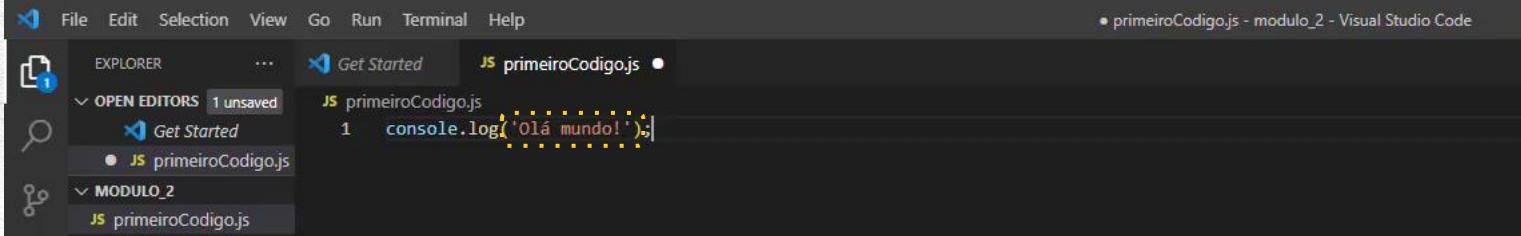
```
File Edit Selection View Go Run Terminal Help
EXPLORER ... Get Started JS primeiroCodigo.js
OPEN EDITORS 1 unsaved JS primeiroCodigo.js
Get Started
primeiroCodigo.js
MODULO_2 JS primeiroCodigo.js
```

primeiroCodigo.js - modulo_2 - Visual Studio Code

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

Normalmente usamos essa técnica de programação para conseguir enxergar o caminho que o nosso código está fazendo, mas entenderemos isso mais à frente.

No JavaScript ao escrever **TEXTOS** devemos colocá-los entre aspas para que o programa entenda que tipo de informação estamos passando.



```
File Edit Selection View Go Run Terminal Help
EXPLORER ... Get Started JS primeiroCodigo.js
OPEN EDITORS 1 unsaved JS primeiroCodigo.js
Get Started
primeiroCodigo.js
MODULO_2 JS primeiroCodigo.js
```

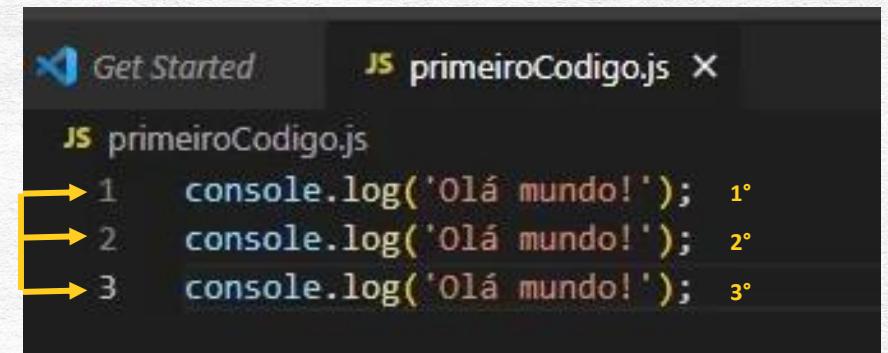
primeiroCodigo.js - modulo_2 - Visual Studio Code

```
1 console.log('Olá mundo!');
```

Javascript é uma linguagem interpretada e leve. O computador recebe o código JavaScript em sua forma de texto original e executa o seu ‘script’ em tempo de execução, e não antes, ou seja, é interpretado/traduzido diretamente no código da linguagem por um interpretador chamado JavaScript Engine. E durante o curso utilizaremos o interpretador **NODEJS**.

2

O código JavaScript é uma sequência de comandos JavaScript. Cada comando é executado pelo computador na sequência que eles são escritos, de cima para baixo. Isso significa que você precisa tomar cuidado com a ordem que você coloca as coisas dentro do seu código.



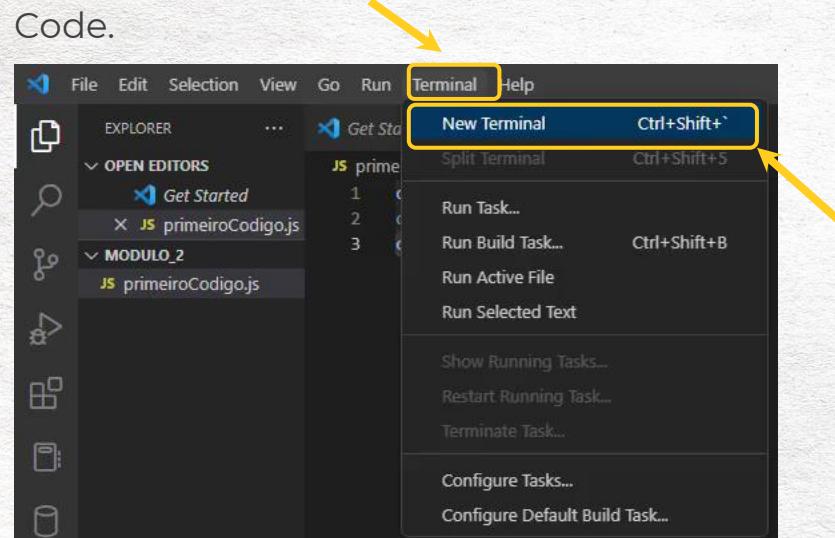
```
Get Started JS primeiroCodigo.js ×  
JS primeiroCodigo.js  
→ 1 console.log('Olá mundo!'); 1°  
→ 2 console.log('Olá mundo!'); 2°  
→ 3 console.log('Olá mundo!'); 3°
```



Para executarmos o Node.js e interpretar nosso código JavaScript iremos realizar as seguintes instruções!

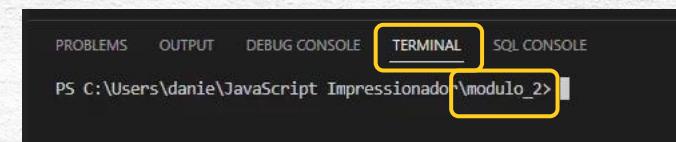
Instrução 1º

Iremos clicar com o botão esquerdo em 'Terminal' e logo em seguida clicar em 'Novo Terminal', para iniciarmos o Terminal de comando dentro do VS Code.



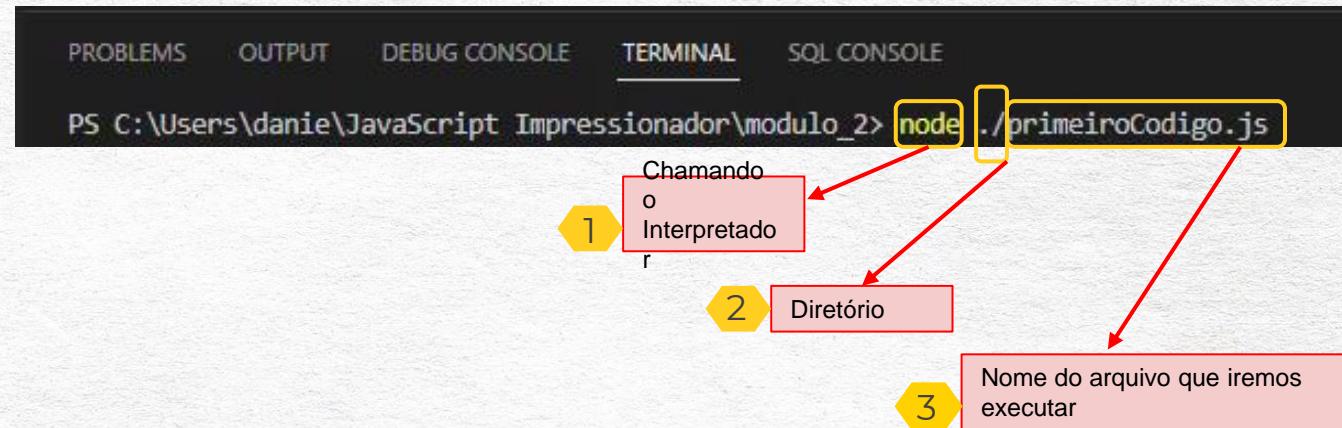
Instrução 2º

Na parte inferior do VS Code, o Terminal irá abrir e apontar para o local / pasta que estamos trabalhando. Nesse caso, no diretório 'modulo_2'. E dentro do terminal poderemos escrever comandos para interpretar e executar o nosso primeiro programa.



Instrução 3º

No terminal iremos digitar o comando **node**, pois será o interpretador que utilizaremos dentro do nosso programa para ler e executar o nosso código. Seguindo de '**./**', que corresponde "a partir do diretório que estou trabalhando", que é o local / pasta que está salvo o nosso arquivo de Javascript. E para finalizar escrevemos o nome do arquivo que queremos executar, que no nosso caso é '**primeiroCodigo.js**'.



```
Get Started JS primeiroCodigo.js X
JS primeiroCodigo.js
1 console.log('Olá mundo!');
2 console.log('Olá mundo!');
3 console.log('Olá mundo!');

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo!
Olá mundo!
Olá mundo!
PS C:\Users\danie\JavaScript Impressionador\modulo_2> []
```

Enter

Instrução 4º

E agora é hora de executarmos o nosso primeiro programa em JavaScript apertando a tecla '**ENTER**' e verificar se a saída é a instrução que queríamos que nosso código fizesse. No nosso programa esperamos que o comando **console.log** IMPRIMA três vezes o texto "Olá mundo!" que escrevemos dentro do seus parênteses () .

"TUDO COMEÇA COM UM 'Olá Mundo!' (ou 'Hello World!')"

Uma curiosidade para você é que o 'Olá Mundo!', é uma frase que foi imortalizada pelos criadores da linguagem de programação C, porém foi aderida por toda a comunidade de programação como uma brincadeira, para iniciarmos com o pé direito em uma linguagem, criamos o nosso primeiro programa dizendo 'Olá Mundo!'.

The screenshot shows a terminal window with the following interface elements:

- File tabs: "Get Started" and "primeiroCodigo.js X".
- Code editor tab: "primeiroCodigo.js".
- Code content:

```
1  console.log('Eu acabei de escrever meu primeiro código JavaScript!!');
```
- Terminal tab: "TERMINAL" (underlined).
- SQL CONSOLE tab.
- Terminal output:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\primeiroCodigo.js
Eu acabei de escrever meu primeiro código JavaScript!!
PS C:\Users\danie\JavaScript Impressionador\modulo_2>
```

Meu Primeiro Programa JavaScript

E agora para finalizarmos iremos trocar o texto do nosso comando `console.log()`, por “Eu acabei de escrever meu primeiro código JavaScript!!” e executamos as mesmas instruções do “Olá mundo!”.

Pronto!! Você acabou de escrever e executar o seu primeiro programa, incrível não!?





Limpar o terminal

Dentro do seu terminal você irá escrever o comando **cls** ou **clear** e apertar a tecla ‘ENTER’ para executar. A tela do seu terminal será totalmente limpa e estará pronta para receber novas instruções.



Caminhar pelo terminal

Ao clicarmos na tecla ‘**para cima**’ e ‘**para baixo**’ no seu teclado, você caminhará através dos comandos que você já executou no seu terminal. São atalhos para facilitar seu dia a dia como programador.

Nesta aula, aprenderemos **duas formas de abrir o VS Code** em seu projeto. Seja você iniciante ou já com alguma experiência, essas são práticas essenciais para configurar corretamente seu ambiente de desenvolvimento.

Forma 1: Abrindo o Último Projeto Trabalhado

Passo a Passo:

- **Abrir o VS Code:**
 - Localize o ícone do Visual Studio Code no seu computador e clique para abri-lo.
 - O VS Code abrirá automaticamente o **último projeto** no qual você trabalhou, desde que:
 - Você não tenha desinstalado o programa.
 - Não tenha excluído ou movido os arquivos do último projeto.
- **Verificando o Projeto Aberto:**
 - Assim que o VS Code carregar, verifique na barra lateral esquerda (ou no topo, dependendo do sistema) se os arquivos e pastas correspondem ao seu último projeto.
 - Caso você queira trabalhar em outro projeto, siga para o próximo passo.
- **Abrindo Outra Pasta (Caso Necessário):**
 - Clique em **File** (Arquivo) no menu superior.
 - Selecione **Open Folder...** (Abrir Pasta).
 - Navegue até a pasta onde está o projeto desejado, selecione-a e clique em **OK** ou **Selecionar Pasta**.

Forma 2: Abrindo o VS Code Sem Nenhuma Conexão de Pasta

Passo a Passo:

- **Abrir o VS Code em Branco:**

- Abra o Visual Studio Code sem se preocupar com o último projeto.
- Ele será carregado sem nenhum projeto ou pasta conectada.

- **Conectar o Projeto:**

- No canto superior esquerdo, clique em **File** (Arquivo).
- Escolha a opção **Open Folder...** (Abrir Pasta).
- Encontre a pasta do seu projeto no navegador de arquivos que será aberto.
- Selecione a pasta e clique em **OK** ou **Selecionar Pasta**.

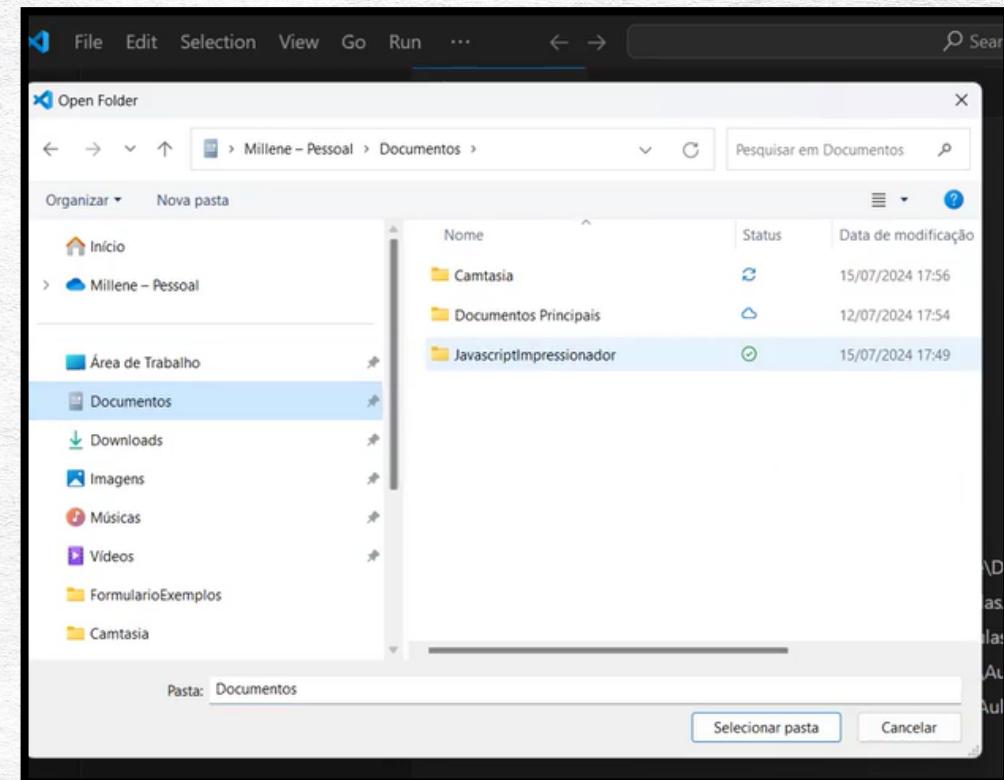
- **Verificar a Conexão:**

- Depois de abrir a pasta, verifique na barra lateral do VS Code se aparecem os arquivos e pastas do projeto.
- Isso confirma que o ambiente está configurado corretamente.

Dica Importante: Salvando Tempo

Para abrir uma pasta diretamente no VS Code usando o terminal:

- Navegue até a pasta do projeto pelo terminal.
- Digite o comando `code .` e pressione **Enter**.
- O VS Code abrirá automaticamente na pasta especificada.





Auto preenchimento

A tecla **Tab** no terminal é uma funcionalidade muito útil que permite **autocompletar nomes de arquivos ou pastas** enquanto você digita comandos. Isso facilita e acelera o processo, especialmente quando os nomes são longos ou complexos.

```
or> node ./meuPrime
```

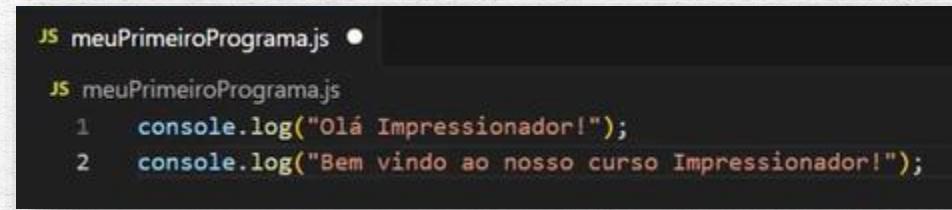
Erro Comum: Não Salvar Alterações no VS Code

Um erro comum entre desenvolvedores, especialmente em IDEs como o **Visual Studio Code**, é **esquecer de salvar as alterações feitas em arquivos**. Isso pode levar a vários problemas, como código não atualizado sendo executado ou mudanças que não são refletidas na aplicação. Vamos entender as causas e como evitá-lo.

Como Evitar Esse Erro

- **Verifique o Status do Arquivo:**

- Sempre que você fizer uma alteração, observe a aba do arquivo para ver se ele está marcado com o asterisco (*) ou com o ponto de exclamação, indicando que há mudanças não salvas.



```
JS meuPrimeiroPrograma.js •  
JS meuPrimeiroPrograma.js  
1 console.log("Olá Impressionador!");  
2 console.log("Bem vindo ao nosso curso Impressionador!");
```

- **Use o Atalho de Salvar Regularmente:**

- Salve frequentemente com **Ctrl + S** (ou **Cmd + S** no Mac) para garantir que suas alterações sejam mantidas.
- Você também pode configurar o VS Code para **salvar automaticamente** ao perder o foco (você pode ativar isso nas configurações de "Auto Save").

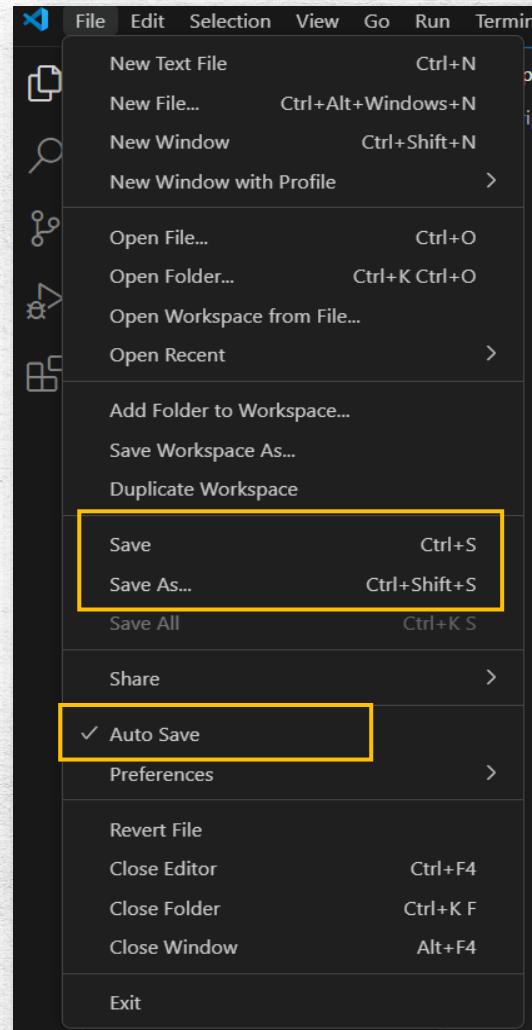
- **Habilite o Auto Save:**

- No VS Code, é possível configurar o **Auto Save**, que salvará seus arquivos automaticamente em intervalos definidos ou sempre que você mudar de foco.



Salvar

Salve frequentemente com **Ctrl + S**
(ou **Cmd + S** no Mac) para garantir
que suas alterações sejam
mantidas.



Auto Save

No VS Code, é possível configurar o **Auto Save**, que salvará seus arquivos automaticamente em intervalos definidos ou sempre que você mudar de foco.

Para ativar, vá até as **Configurações** e pesquise por "Auto Save"

Erro Comum: **Escrever o Comando node Junto com o Arquivo, Tratando Tudo Como um Único Comando**

Esse é um erro comum que ocorre quando você escreve o comando node de maneira incorreta no terminal. Isso pode resultar em uma execução inesperada ou erro ao tentar rodar o seu arquivo JavaScript.

O Que Acontece Quando Isso Acontece?

Quando você escreve o comando node seguido de um arquivo (e talvez de alguns outros caracteres ou espaços adicionais), o terminal pode interpretar isso de maneira incorreta, fazendo com que o comando **não seja reconhecido** corretamente. Isso acontece quando você junta o nome do comando com o arquivo de forma que o terminal interprete a linha toda como um comando, ao invés de apenas o **comando node** e o **arquivo** que você deseja executar.

```
PS C:\Users\mikag\OneDrive\Documentos\JavascriptImpressionador> node ./meuPrimeiroPrograma.js
```

```
node./meuPrimeiroPrograma.js : O termo 'node./meuPrimeiroPrograma.js' não é reconhecido como nome de cmdlet, função, arquivo de script ou programa operável. Verifique a
grafia do nome ou, se um caminho tiver sido incluído, veja se o caminho está correto e tente novamente.
No linha:1 caractere:1
+ node./meuPrimeiroPrograma.js
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (node./meuPrimeiroPrograma.js:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

A correção é simples: certifique-se de **separar o comando e o arquivo corretamente**, incluindo apenas o nome do arquivo que você deseja executar após o comando node.

Dicas Adicionais

- **Verifique o Caminho do Arquivo:** Se o arquivo não estiver no diretório atual, você precisará informar o **caminho completo** ou relativo para ele.
- **Múltiplos Arquivos com node:** O comando node não permite a execução de múltiplos arquivos ao mesmo tempo (ao contrário de alguns outros ambientes, como o python). Você precisa executar um arquivo por vez.
- **Tab para Autocompletar:** Se você tiver dificuldades em digitar o nome completo do arquivo, pode usar a tecla **Tab** para autocompletar o nome do arquivo.

Quando for usar o comando node, lembre-se de sempre **separar o comando do nome do arquivo**. Dessa forma, o terminal conseguirá interpretar corretamente o que você deseja executar. Ao evitar esse erro comum, você terá um fluxo de trabalho mais tranquilo e produtivo!

Erro Comum: **Escrever o Nome do Arquivo Errado ao Usar o Comando node**

Outro erro comum ao trabalhar com o comando node no terminal é **escrever o nome do arquivo errado**. Isso pode causar um erro que impede a execução do seu código, porque o terminal não consegue encontrar o arquivo especificado. Vamos entender o que acontece e como evitar esse erro.

O Que Acontece Quando Você Escreve o Nome do Arquivo Errado?

Quando você tenta rodar um arquivo com o comando node e digita o nome errado, o terminal não encontra o arquivo e retorna um **erro de "arquivo não encontrado"** ou algo semelhante. Isso acontece porque o terminal tenta localizar o arquivo com o nome exato que você forneceu, e, se o arquivo não existir ou o nome estiver incorreto, ele não conseguirá executá-lo.

Exemplo de Erro:

```
node:internal/modules/cjs/loader:1147
  throw err;
  ^

Error: Cannot find module 'C:\Users\Edição 4\Documents\Plantão Dúvidas\Rogério\meuPrimeiroProgra.js'
  at Module._resolveFilename (node:internal/modules/cjs/loader:1144:15)
  at Module._load (node:internal/modules/cjs/loader:985:27)
  at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:135:12)
  at node:internal/main/run_main_module:28:49 {
  code: 'MODULE_NOT_FOUND',
  requireStack: []
}

Node.js v20.11.0
```

- O Node.js tenta localizar o módulo (arquivo) chamado meuprimeiroProgra.js, mas não o encontra no diretório atual.
- O erro "**Cannot find module**" indica que o arquivo não foi encontrado.



Erro Comum: Estar em uma Pasta Errada ao Executar o Comando node

- Outro erro comum ao tentar executar um arquivo com node ocorre quando você está **em uma pasta errada** no terminal. Isso significa que, embora você tenha o comando correto e o nome do arquivo correto, o terminal não consegue encontrar o arquivo porque o **diretório atual não é o diretório correto** onde o arquivo está localizado.

O Que Acontece Quando Você Está na Pasta Errada?

Quando você tenta executar um arquivo com node e está em uma pasta diferente de onde o arquivo está, o terminal não conseguirá localizá-lo. O erro retornado será similar ao erro de nome de arquivo incorreto, já que o sistema não consegue encontrar o arquivo no diretório em que você está.

O Node.js tenta localizar meuPrimeiroPrograma.js no diretório **atualmente ativo** (a pasta novaPasta), mas não encontra o arquivo lá, então ele retorna o erro "**Cannot find module**".

The screenshot shows a terminal window with a file tree on the left and a code editor on the right. The file tree shows a directory structure under 'JAVASCRIPTIMPRESSIONADOR': 'novaPasta' contains 'script.js' and 'meuPrimeiroPrograma.js'. The 'script.js' file is selected. The code editor on the right shows the contents of 'script.js':

```
novaPasta > JS script.js
1 console.log("Estamos dentro de uma outra pasta");
2
```

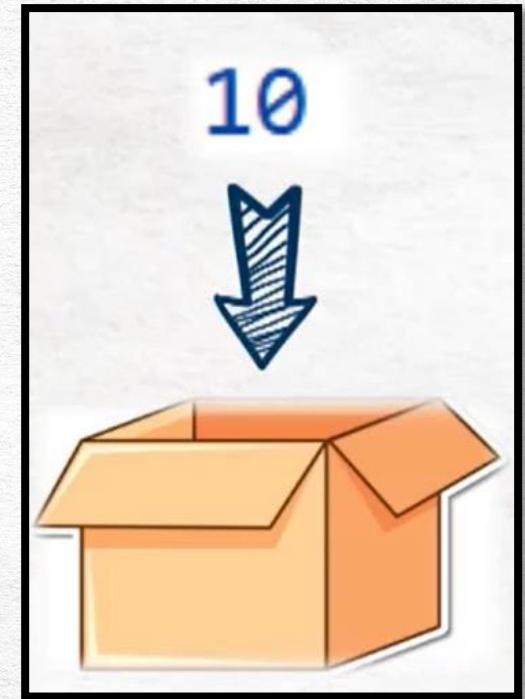
O que são Variáveis em JavaScript?

Em **JavaScript**, uma **variável** é um **espaço na memória** onde podemos armazenar dados, como números, strings (texto), objetos, entre outros. Quando declaramos uma variável, estamos dizendo ao programa que queremos guardar um valor que poderá ser usado e manipulado ao longo da execução do código. Uma variável pode ser alterada ou atualizada durante a execução do programa.

As variáveis são essenciais para a programação porque permitem armazenar informações de forma dinâmica, ou seja, você pode modificar o valor armazenado conforme o código vai sendo executado.

A Evolução das Variáveis em JavaScript

JavaScript, ao longo do tempo, passou por várias mudanças significativas na forma como variáveis são declaradas. As principais mudanças ocorreram com a introdução de novas palavras-chave para declarar variáveis, como var, let e const, cada uma com características e comportamentos específicos.



1. var (1995-2015)

Quando o JavaScript foi criado em 1995, a única maneira de declarar variáveis era usando o **var**. Esse era o método padrão para criar variáveis até 2015.

Características do var:

- **Escopo de função:** As variáveis declaradas com var têm escopo de função, o que significa que elas estão disponíveis dentro da função em que foram declaradas, mesmo que você as declare dentro de um bloco (como um if ou for). Isso pode causar comportamentos inesperados e difíceis de rastrear.
- **Hoisting:** Variáveis declaradas com var são "elevadas" (hoisted) ao topo da função ou do escopo global, o que significa que elas podem ser usadas antes de sua declaração, mas o valor delas será undefined até que a linha de código em que foram atribuídas seja executada.

2. let (2015 em diante)

Em 2015, com o lançamento do **ES6 (ECMAScript 2015)**, uma nova forma de declarar variáveis foi introduzida: **let**.

Características do let:

- **Escopo de bloco:** Ao contrário do var, o let tem **escopo de bloco**, ou seja, a variável só está disponível dentro do bloco {} onde foi declarada, como em loops e condicionais. Isso torna o código mais previsível e seguro, evitando problemas com o escopo de função.
- **Não há hoisting:** Embora o let também seja "elevado" ao topo do escopo (hoisted), a variável não pode ser acessada antes de sua declaração. Isso significa que, ao tentar acessar uma variável let antes de sua declaração, você recebe um erro de referência.

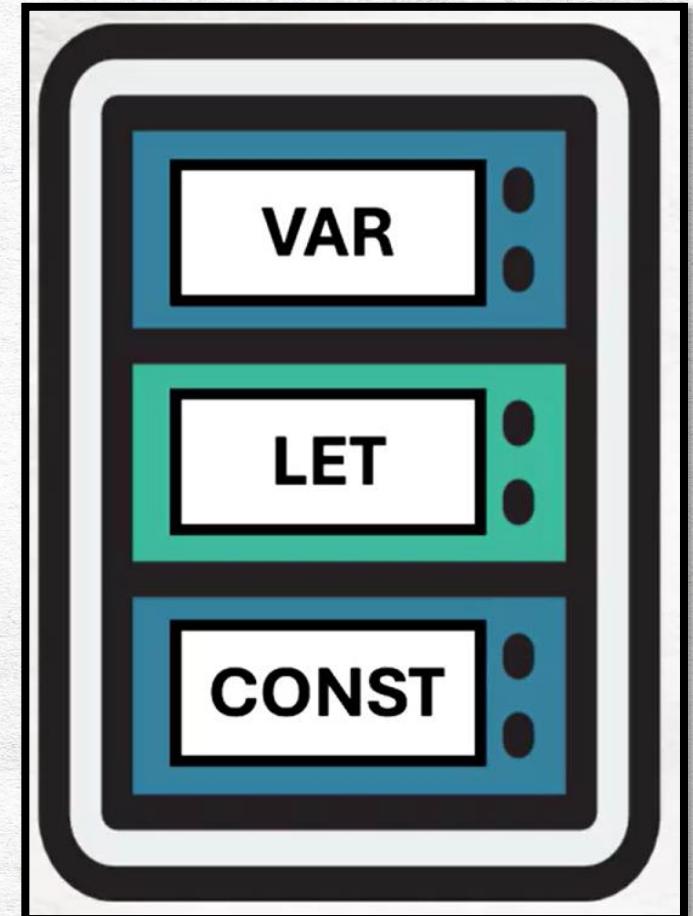
3. const (2015 em diante)

Outra mudança importante com o **ES6** foi a introdução de **const**, uma forma de declarar variáveis que não podem ser reatribuídas após a declaração. O const é usado quando você sabe que o valor da variável não vai mudar ao longo da execução do código.

Características do const:

- **Escopo de bloco:** Assim como o let, o const tem escopo de bloco.
- **Imutabilidade:** Uma vez que uma variável é declarada com const, seu valor **não pode ser reatribuído**. Isso ajuda a garantir que valores importantes, como configurações ou chaves de objetos, não sejam alterados acidentalmente.

A introdução de let e const no ES6 trouxe grandes melhorias para o JavaScript, ajudando a tornar o código mais claro, previsível e livre de erros inesperados relacionados ao escopo. **var** ainda é utilizado em muitos projetos legados, mas a recomendação é usar **let** quando for necessário reatribuir valores e **const** para garantir a imutabilidade da variável, tornando o código mais seguro e fácil de entender.



Nesta aula, vamos aprender o que é uma **declaração de variável** em JavaScript, como ela funciona e qual é a **sintaxe básica** para declarar variáveis.

O que é uma Variável em JavaScript?

Em programação, uma **variável** é um espaço na memória do computador que armazena dados, como números, strings (texto), objetos ou outros tipos de informação. Em JavaScript, você usa variáveis para armazenar e manipular dados ao longo da execução do seu código.

Por Que Usamos Variáveis?

Usamos variáveis para:

- Armazenar dados temporários.
- Manipular esses dados durante a execução do código.
- Tornar o código mais flexível e dinâmico, pois podemos alterar o valor da variável durante a execução.

Como Declarar uma Variável em JavaScript?

Em JavaScript, podemos declarar variáveis usando três palavras-chave: var, let e const. Cada uma tem características específicas, mas todas servem para o mesmo propósito de armazenar dados.

Sintaxe Básica de Declaração de Variáveis:

```
let nomeDaVariavel = valor;
```

- **let**: Palavra-chave para declarar a variável (pode ser var ou const dependendo do contexto).
- **nomeDaVariavel**: O nome da variável que estamos criando. Ele pode ser qualquer sequência de caracteres, mas deve seguir algumas regras (não pode começar com número, não pode ter espaços, etc.).
- **=**: O sinal de igual é o operador de **atribuição**, ou seja, ele atribui um valor à variável.
- **valor**: O valor que será armazenado na variável (pode ser um número, texto, array, etc.)

```
let idade = 25;           // Variável do tipo número
let nome = "João";       // Variável do tipo string (texto)
let ativo = true;         // Variável do tipo booleano
```

Reatribuições de Valores em Variáveis no JavaScript

Em JavaScript, a **reatribuição de valor** acontece quando você altera o valor de uma variável depois que ela foi declarada. O processo é simples: você pode usar o **operador de atribuição =** para dar um novo valor à variável. Contudo, a forma como as reatribuições funcionam depende da palavra-chave usada para declarar a variável, como var, let ou const. Vamos explorar cada um desses casos.

Reatribuição de Valores com let e

Tanto o let quanto o var permitem que você **reatribua valores** às variáveis após a declaração. Isso significa que, se você declarar uma variável com let ou var, poderá alterar seu valor mais de uma vez durante a execução do programa.

```
let idade = 30; // Atribuição inicial  
console.log(idade); // 30  
  
idade = 35; // Reatribuição  
console.log(idade); // 35
```

Neste exemplo, a variável idade foi inicialmente atribuída com o valor 30, mas depois foi reatribuída para 35. A reatribuição é feita usando o operador de atribuição =.

Reatribuição de Valores com const

Ao contrário de let e var, **const** é usado para declarar **variáveis imutáveis**. Isso significa que, após você atribuir um valor a uma variável declarada com const, **não é possível reatribuir um novo valor** a essa variável. Qualquer tentativa de reatribuição resultará em um erro.

```
const pais = "Brasil"; // Atribuição inicial
console.log(pais);    // Brasil

// Tentando reatribuir
pais = "Argentina"; // Erro! Não podemos reatribuir um valor a uma variável 'const'
```

No exemplo acima, tentamos alterar o valor da variável pais que foi declarada com const. O Javascript geraria um erro como:

```
segundaMensagem = "Quero trocar a mensagem da minha variável";
^

TypeError: Assignment to constant variable.
```

Isso ocorre porque const **não permite reatribuição**.



Imutabilidade com const

Embora você não possa reatribuir uma variável const, **isso não significa que o valor armazenado seja imutável**, especialmente quando o valor é um **objeto** ou um **array**. Embora você não possa **reatribuir** o objeto ou array, você pode modificar seu conteúdo, como adicionar ou remover propriedades de um objeto ou elementos de um array.

Diferença de Reatribuição entre let, var e const

Característica	let	var	const
Reatribuição	Permite reatribuição	Permite reatribuição	Não permite reatribuição
Escopo	Escopo de bloco (dentro de um bloco de código)	Escopo de função (dentro de uma função)	Escopo de bloco (dentro de um bloco de código)
Hoisting	Sim, mas com "temporal dead zone" (não pode ser acessado antes da declaração)	Sim, mas inicializada com undefined	Sim, mas não pode ser acessada antes da declaração

- **let e var** permitem que você reatribua valores às variáveis ao longo do código. A principal diferença entre os dois é o escopo. Enquanto let tem escopo de bloco, var tem escopo de função, o que pode gerar confusão em situações mais complexas.
- **const** é usado quando você deseja que a variável **não seja reatribuída**. No entanto, se o valor armazenado for um objeto ou array, você ainda pode modificar suas propriedades ou elementos, mas não pode atribuir um novo valor para a variável.

A escolha entre let, var e const deve ser feita com base na necessidade de mutabilidade ou imutabilidade da variável. É uma boa prática usar const por padrão e let quando a variável precisa ser alterada ao longo do tempo.

Explicação sobre o var em JavaScript e como ele funciona

Em JavaScript, o var é uma das palavras-chave usadas para **declarar variáveis**. Ele foi o método principal de declaração de variáveis nas primeiras versões da linguagem, antes da introdução do let e const no ECMAScript 6 (ES6) em 2015.

Características do var:

- **Escopo de função:** O var tem escopo de função, o que significa que, quando você declara uma variável com var dentro de uma função, ela só será acessível dentro dessa função. Se declarada fora de uma função, ela se torna uma variável global.
- **Hoisting:** O var sofre o comportamento de **hoisting**, que significa que a variável é "movida" para o topo do seu escopo de execução (função ou global). Isso pode causar confusão, porque você pode acessar uma variável declarada com var antes mesmo de sua linha de declaração, mas ela será inicializada com o valor undefined até a linha onde foi atribuída.

```
console.log(nome); // undefined (não gera erro, mas o valor é undefined)
var nome = "João";
console.log(nome); // João
```



- **Redeclaração:** No escopo de função ou global, você pode **redeclara** uma variável com var sem causar erro. Isso pode levar a resultados inesperados se você redeclarar uma variável sem perceber.

```
var nome = "Lucas";
var nome = "Carlos"; // Sem erro, redeclaração permitida
console.log(nome); // Carlos
```

- **Escopo global:** Quando você declara uma variável fora de qualquer função com var, ela se torna uma variável global e pode ser acessada de qualquer parte do código.

E quando você declara uma variável sem tipo explícito?

- Em JavaScript, **não existe um tipo explícito de variável**, porque a linguagem é **dinamicamente tipada**. Isso significa que o tipo da variável é determinado automaticamente com base no valor atribuído a ela. **Não é necessário especificar se a variável será um número, string, booleano, etc..**
- Se você declarar uma variável sem usar palavras-chave como var, let ou const, o JavaScript automaticamente a entenderá como uma variável global, e esse comportamento é muito parecido com o do var.

```
nome = "Maria"; // Declarando sem var, let ou const
console.log(nome); // Maria
```



Neste exemplo, a variável nome é automaticamente **global** e **equivalente a uma variável declarada com var**. Isso pode levar a problemas, pois não há controle sobre o escopo da variável, e ela pode ser sobrescrita accidentalmente em qualquer parte do código.

• **Importante:** Declarar variáveis dessa forma (sem usar var, let ou const) não é uma boa prática. Embora o JavaScript permita, isso pode tornar o código difícil de depurar e mais propenso a erros. **Sempre use let, const ou var** para declarar variáveis explicitamente, o que melhora a legibilidade e segurança do código.

Exemplo com var, let e const:

```
var nome = "Pedro"; // Declaração com var  
let idade = 28;     // Declaração com let  
const cidade = "Rio"; // Declaração com const  
  
console.log(nome, idade, cidade); // Pedro 28 Rio
```

- **var** é uma palavra-chave usada para declarar variáveis e possui escopo de função (ou global, se declarada fora de funções). Além disso, a variável declarada com var sofre hoisting e permite redeclaração.
- Se você **declarar uma variável sem tipo explícito**, o JavaScript **não cria automaticamente uma variável com var**, mas sim uma variável global (sem a palavra-chave). Isso pode ser arriscado, pois a variável fica fora de controle, podendo ser modificada em qualquer parte do código, o que pode causar problemas.
- **Boas práticas:** Use **sempre** let ou const para declarar variáveis, pois eles têm escopo de bloco e evitam problemas com hoisting e redeclaração. Evite declarar variáveis sem tipo, para garantir que o código se mantenha organizado e seguro.

Diferenças entre var, let e const em JavaScript

Em JavaScript, temos três palavras-chave principais para declarar variáveis: **var**, **let** e **const**. Cada uma tem características distintas que afetam o comportamento da variável, seu escopo, e como ela pode ser manipulada. A seguir, vamos explorar essas diferenças de forma detalhada.

1. Escopo (Onde a variável pode ser acessada)

- **var**:

Escopo de função: Quando você declara uma variável com var dentro de uma função, ela fica acessível apenas dentro dessa função. Porém, se você a declarar fora de qualquer função, ela se torna uma **variável global**.

```
function exemploVar() {  
    var nome = "João"; // 'nome' tem escopo de função  
}  
console.log(nome); // Erro! 'nome' não está acessível fora da função
```

- **let**:

Escopo de bloco: A variável declarada com let tem um **escopo de bloco**, ou seja, ela está disponível apenas dentro do bloco de código (como dentro de um if, for, ou dentro de uma função).

```
if (true) {  
    let nome = "Lucas"; // 'nome' está dentro do bloco do if  
    console.log(nome); // Lucas  
}  
console.log(nome); // Erro! 'nome' não está acessível fora do bloco
```



- **const:**

Escopo de bloco: Assim como let, a variável declarada com const também possui **escopo de bloco**.

```
if (true) {  
  const idade = 25; // 'idade' tem escopo de bloco  
  console.log(idade); // 25  
}  
console.log(idade); // Erro! 'idade' não está acessível fora do bloco
```

2. Reatribuição de Valores (Alterar o valor da variável)

- **var:**

Permite reatribuição: Com var, você pode alterar o valor da variável depois de ela ser declarada sem problemas.

```
var nome = "João";  
console.log(nome); // João  
nome = "Carlos";  
console.log(nome); // Carlos
```



- **let:**

Permite reatribuição: Assim como var, a variável declarada com let pode ser reatribuída com um novo valor.

```
let idade = 30;
console.log(idade); // 30
idade = 35;
console.log(idade); // 35
```

- **const:**

Não permite reatribuição: Variáveis declaradas com const não podem ser reatribuídas após a declaração. Isso faz delas variáveis imutáveis.

```
const cidade = "São Paulo";
console.log(cidade); // São Paulo
cidade = "Rio de Janeiro"; // Erro! Não pode reatribuir
```

Importante: Embora você não possa reatribuir um valor à variável const, se o valor da variável for um **objeto ou array**, você ainda pode **alterar suas propriedades** ou elementos. O que não pode ser feito é **reatribuir** o objeto ou array inteiro.

3. Hoisting (Comportamento de elevação da declaração para o topo)

- **var:**

O var sofre **hoisting**, o que significa que a declaração da variável é "elevada" para o topo do escopo, mas a atribuição de valor **não** é. Ou seja, você pode acessar a variável antes da linha de declaração, mas ela terá o valor undefined até a linha onde for atribuída um valor.

```
console.log(nome); // undefined (não gera erro, mas o valor é undefined)
var nome = "João";
console.log(nome); // João
```

- **let:**

O let também sofre **hoisting**, mas ao contrário de var, ele possui o conceito de **temporal dead zone (TDZ)**. Ou seja, você **não pode acessar** a variável antes da linha de declaração, e isso causará um erro.

```
console.log(nome); // Erro! 'nome' não pode ser acessado antes da declaração
let nome = "Maria";
```

- **const:**

O const também sofre **hoisting** e possui o comportamento de **temporal dead zone (TDZ)**. Assim como com let, você não pode acessar a variável antes de sua declaração.

```
console.log(nome); // Erro! 'nome' não pode ser acessado antes da declaração  
const nome = "Ana";
```

4. Redeclaração (Declarar novamente a mesma variável)

- **var:**

- **Permite redeclaração:** Dentro do mesmo escopo (função ou global), você pode redeclarar a variável declarada com var sem gerar erro.

```
var nome = "Lucas";  
var nome = "Carlos"; // Não gera erro, redeclaração permitida  
console.log(nome); // Carlos
```

- **let:**

Não permite redeclaração: Dentro do mesmo escopo, não é possível redeclarar uma variável já declarada com let. Isso causará um erro.

```
let nome = "Lucas";
let nome = "Carlos"; // Erro! Não é permitido redeclarar uma variável com 'let'
```

- **const:**

Não permite redeclaração: Assim como let, você não pode redeclarar uma variável já declarada com const.

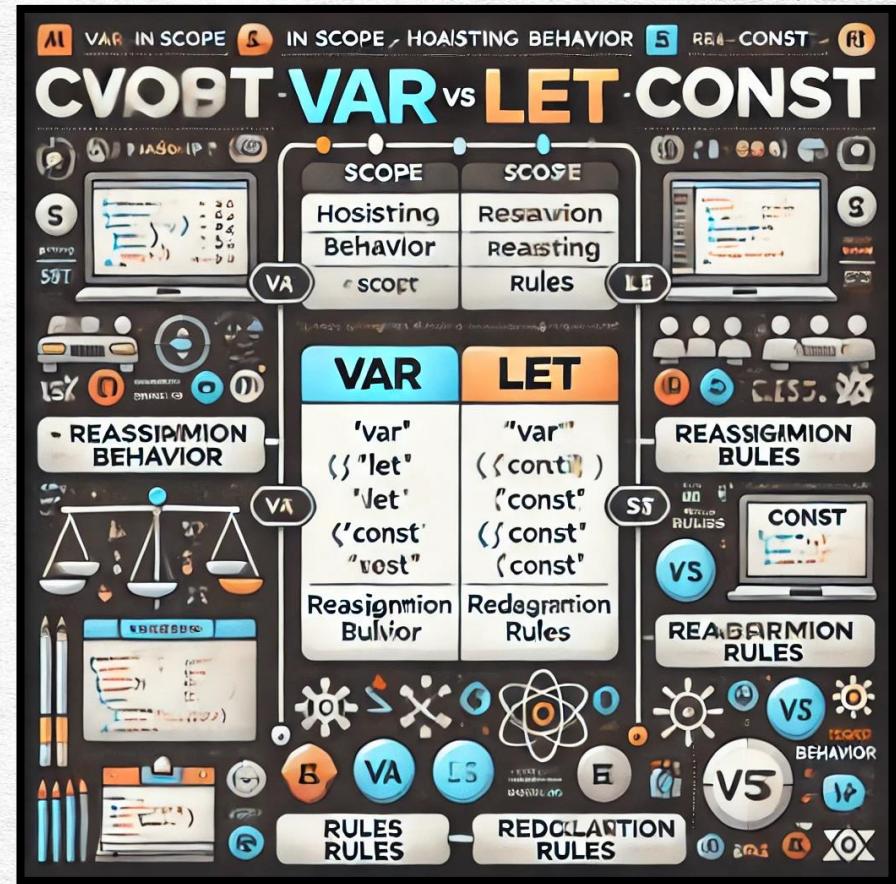
```
const nome = "Lucas";
const nome = "Carlos"; // Erro! Não é permitido redeclarar uma variável
```

Tabela Resumo:

Característica	var	let	const
Escopo	Escopo de função ou global	Escopo de bloco	Escopo de bloco
Reatribuição	Permite reatribuição	Permite reatribuição	Não permite reatribuição
Hoisting	Sim, mas inicializada com <code>undefined</code>	Sim, com temporal dead zone (TDZ)	Sim, com temporal dead zone (TDZ)
Redeclaração	Permite redeclaração no mesmo escopo	Não permite redeclaração	Não permite redeclaração

Conclusão:

- **Var** é a palavra-chave mais antiga para declarar variáveis, mas tem um escopo de função e sofre hoisting, o que pode causar erros difíceis de depurar. Não é mais recomendado para uso em código moderno.
- **let** e **const** foram introduzidos no ES6 (2015) e têm escopo de bloco, o que torna o código mais previsível e seguro. **let** permite reatribuição, enquanto **const** cria variáveis imutáveis (embora você possa modificar objetos e arrays declarados com **const**).
- A melhor prática é usar **const** por padrão (para garantir imutabilidade) e **let** quando for necessário reatribuir valores.
- Essas diferenças são essenciais para escrever código mais seguro e fácil de manter.



Regras e Boas Práticas de Nomenclatura em JavaScript

Manter um código limpo e compreensível é essencial para a manutenção e evolução de projetos de programação. Quando falamos sobre boas práticas em JavaScript, um dos pontos mais importantes é a **nomenclatura** de variáveis e funções. Vamos explorar algumas regras e recomendações que ajudam a manter a clareza e consistência no código.

1. Nomes Significativos

Sempre utilize **nomes descritivos** para variáveis e funções. Um nome significativo deve refletir o propósito do valor armazenado ou da ação realizada pela função. Por exemplo, ao invés de usar um nome genérico como x, prefira algo como contagemDeltens ou totalDeProdutos para facilitar o entendimento de quem estiver lendo o código.

2. Notação Camel Case

No JavaScript, é comum utilizar a **notação camelCase** para nomear variáveis e funções. Isso significa que o nome começa com a primeira palavra em minúscula e as palavras subsequentes começam com letras maiúsculas, sem espaços ou caracteres especiais.

Por exemplo:

- **variavelDeContagem** (em vez de variavel_de_contagem ou VariavelDeContagem).

Essa convenção é importante para manter o código legível e consistente, especialmente em projetos colaborativos.

3. Evitar Palavras Reservadas

Existem **palavras reservadas** no JavaScript, que têm um significado especial para a linguagem. Tais palavras não podem ser usadas como nomes de variáveis, funções ou outros identificadores. Alguns exemplos de palavras reservadas incluem: for, if, class, return. Usar essas palavras pode gerar erros inesperados ou comportamentos errados no seu código.

4. Não Começar com Número

Em JavaScript, **variáveis não podem começar com números**. Elas devem iniciar com uma letra (maiúscula ou minúscula), um caractere de sublinhado (_) ou o símbolo do dólar (\$). Por exemplo, 2var é inválido, mas var2 é perfeitamente aceitável.

5. Camel Case vs Snake Case

Existem diferentes convenções para separar palavras em nomes de variáveis. A **notação Camel Case** é a mais comum em JavaScript, mas em alguns casos, a **notação Snake Case** pode ser usada. Na notação Snake Case, as palavras são separadas por um sublinhado (_) e todas as letras são minúsculas, como em variavel_de_contagem.

É importante manter a consistência em todo o projeto, optando por uma convenção e aplicando-a de maneira uniforme.

6. Aspas Simples vs Aspas Duplas

Em JavaScript, você pode escolher entre **aspas simples** ('') ou **aspas duplas** ("") para delimitar strings. A recomendação aqui é **manter a consistência** no seu código. Embora não exista uma regra rígida, aspas simples são geralmente mais usadas, a menos que seja necessário usar aspas duplas dentro de uma string, como em um texto que já contenha aspas simples. Exemplo:

- Aspas Simples: 'Olá, Mundo!'
- Aspas Duplas: "A variável 'nome' foi alterada"

7. Uso do Ponto e Vírgula

Embora o JavaScript permita que você **omite o ponto e vírgula** em muitas situações, isso pode causar comportamentos inesperados devido às regras de inserção automática de ponto e vírgula. Por isso, é **recomendado** sempre colocar o ponto e vírgula ao final de uma declaração, garantindo que o código se comporte como esperado.

8. Constantes

Quando definir **constants**, utilize **letas maiúsculas** e separe as palavras por **underscores** (_). Isso ajuda a diferenciar constantes de variáveis regulares e melhora a legibilidade do código. Exemplo:

- **const MAX_LIMIT = 100;**

Essas práticas não apenas tornam o código mais organizado, mas também facilitam a colaboração em equipe e a manutenção a longo prazo.

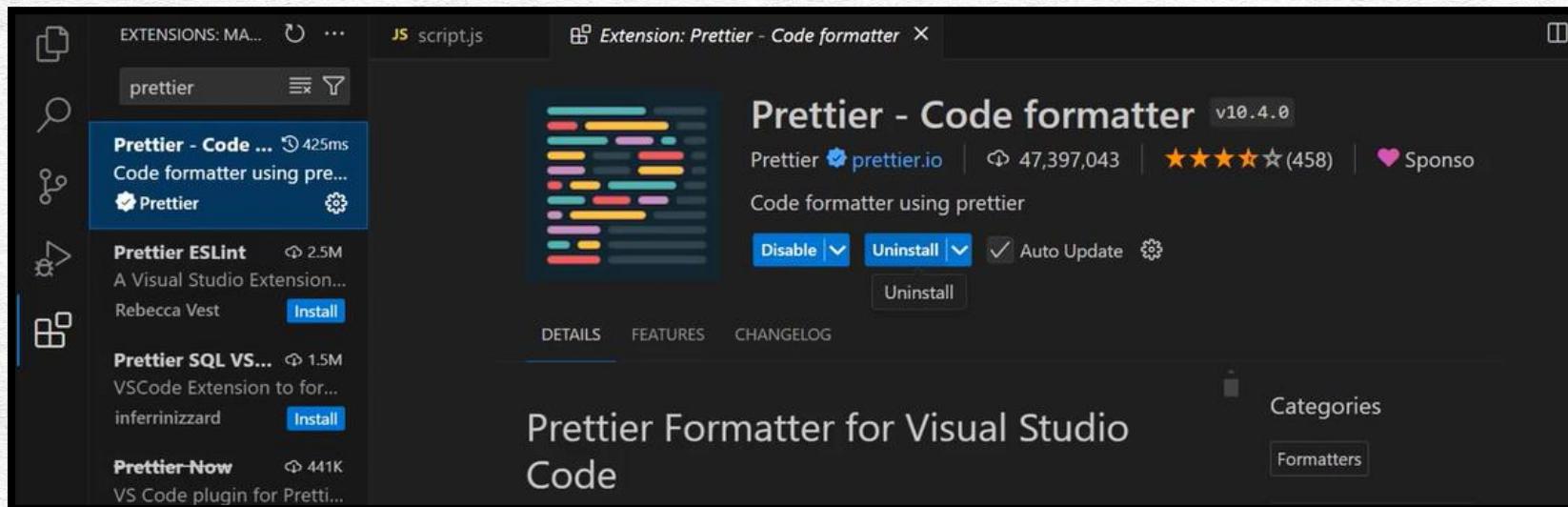
Adotar essas regras e boas práticas ao nomear variáveis e funções em seu código JavaScript é fundamental para criar projetos mais eficientes, claros e de fácil manutenção.

O que é o Prettier?

O **Prettier** é uma ferramenta de formatação automática de código. Ele ajuda a manter um estilo de código consistente e legível, aplicando regras específicas para espaçamento, indentação, quebras de linha, entre outras. Isso é muito útil em projetos colaborativos, onde várias pessoas podem escrever código de maneiras diferentes. Com o Prettier, a formatação é aplicada automaticamente, evitando conflitos e divergências entre estilos.

Por que usar o Prettier no VS Code?

Ao trabalhar com JavaScript (e outras linguagens), a formatação do código é crucial para garantir que o código seja legível e fácil de entender. O Prettier faz isso de maneira automática, economizando tempo e evitando erros comuns de formatação. Além disso, ele ajuda a manter o código limpo e consistente, seguindo as boas práticas de indentação e organização.



Passo 1: Instalando o Prettier no VS Code

- **Abra o VS Code.**
 - **Acesse o Marketplace de Extensões:**
 - No menu lateral esquerdo, clique no ícone de "Extensões" (ou pressione Ctrl + Shift + X).
 - **Busque pelo Prettier:**
 - No campo de busca, digite "Prettier - Code formatter".
 - **Instale a Extensão:**
 - Clique no botão de **Instalar** ao lado da extensão Prettier.
- Agora, o Prettier está instalado no seu VS Code e pronto para ser configurado!

Passo 2: Configurando o Prettier no VS Code

Após instalar o Prettier, vamos configurar algumas opções para garantir que ele formate o código da maneira que você prefere.

- **Configurações Padrão:**
 - O Prettier já tem configurações padrão para formatação de código, mas você pode personalizar essas configurações.
- **Acesse as Configurações:**
 - Clique no ícone de engrenagem no canto inferior esquerdo do VS Code e selecione **Configurações**.
 - Na barra de pesquisa, digite "Prettier" para ver as opções disponíveis.
- **Configurações Importantes:**
 - **Format on Save:** Ative a opção **Editor: Format On Save**. Isso fará com que o Prettier formate automaticamente seu código sempre que você salvar o arquivo.
 - **Prettier Config File:** Você também pode criar um arquivo `.prettierrc` na raiz do seu projeto para personalizar as configurações de formatação, como o estilo de indentação (espaços ou tabulação) ou a largura das linhas.

Passo 3: Usando o Prettier para Formatar Código JavaScript

Agora que o Prettier está instalado e configurado, vamos aprender como utilizá-lo.

1. Formatação Automática ao Salvar o Arquivo

Com a opção **Format On Save** ativada, o Prettier irá formatar automaticamente seu código sempre que você salvar o arquivo. Para salvar o arquivo, basta pressionar Ctrl + S (Windows) ou Cmd + S (Mac).

2. Formatação Manual

Se você preferir formatar o código manualmente, pode fazer isso de duas maneiras:

- **Atalho de Teclado:**
 - No VS Code, pressione Shift + Alt + F (Windows/Linux) ou Shift + Option + F (Mac) para formatar o arquivo atual.
- **Comando de Paleta:**
 - Pressione Ctrl + Shift + P (Windows) ou Cmd + Shift + P (Mac) para abrir a paleta de comandos.
 - Digite "Format Document" e pressione Enter.

3. Verificando a Formatação

Após a formatação, seu código deve estar organizado da seguinte forma:

- Indentação consistente.
- Linhas não muito longas (geralmente 80 ou 120 caracteres).
- Espaços e quebras de linha apropriadas.

Exemplo de código antes e depois da formatação:

Antes (não formatado):

```
function helloWorld(){let name="John";if(name=="John"){console.log("Hello John!");}else{console.log("Hello stranger!");}}
```

Depois (formatado pelo Prettier):

```
function helloWorld() {  
  let name = "John";  
  if (name == "John") {  
    console.log("Hello John!");  
  } else {  
    console.log("Hello stranger!");  
  }  
}
```

Passo 4: Personalizando as Configurações do Prettier

Se você quiser personalizar as regras de formatação do Prettier, crie um arquivo de configuração `.prettierrc` na raiz do seu projeto. Você pode adicionar regras como a largura da linha, se vai usar espaços ou tabulação, e muito mais.

Exemplo de arquivo `.prettierrc`:

```
{  
  "semi": false,  
  "singleQuote": true,  
  "tabWidth": 2  
}
```

Este arquivo configura o Prettier para:

- **Não usar ponto e vírgula** no final das linhas (`semi: false`).
- **Usar aspas simples** ao invés de aspas duplas (`singleQuote: true`).
- **Usar 2 espaços** para a indentação (`tabWidth: 2`).

Conclusão

O Prettier é uma ferramenta poderosa para garantir que seu código esteja sempre bem formatado e legível. Ao instalar e configurar o Prettier no VS Code, você pode automatizar a formatação do seu código, o que economiza tempo e ajuda a evitar erros de formatação. Além disso, personalizando as configurações, você pode garantir que o estilo de código siga as convenções do seu time ou projeto.

Com essa configuração, seu código JavaScript ficará sempre limpo, legível e fácil de manter!



No desenvolvimento de aplicações em JavaScript, entender os **tipos de dados** fundamentais é essencial para escrever código eficiente e livre de erros. Os **tipos primitivos** são os blocos de construção da linguagem e são usados para armazenar e manipular dados simples, como números, texto e valores lógicos.

Neste módulo, vamos explorar os **tipos primitivos** de JavaScript, como:

- **Números** (number),
- **Strings** (string),
- **Booleanos** (boolean),
- **Undefined** (undefined),
- **Null** (null),

Além de entender a teoria por trás desses tipos de dados, também veremos suas **aplicações práticas** no cotidiano da programação. Como você pode usá-los corretamente, como manipulá-los e como evitar erros comuns ao trabalhar com cada tipo. Compreender esses fundamentos é o primeiro passo para se tornar um desenvolvedor JavaScript competente e produzir código robusto e sem surpresas.

Vamos dar o primeiro passo na exploração desses tipos e entender como cada um pode ser utilizado para construir aplicações mais eficientes e legíveis.

No JavaScript, **strings** são usadas para representar dados textuais, ou seja, sequências de caracteres. Elas são essenciais para qualquer aplicação, seja para mostrar mensagens na tela, armazenar nomes de usuários, ou até mesmo manipular dados em arquivos. No entanto, o JavaScript oferece diferentes maneiras de criar e manipular strings, com algumas diferenças sutis entre elas. Vamos explorar as três principais formas de criar strings em JavaScript: **aspas simples ('')**, **aspas duplas ("")**, e **template literals** (ou **template strings**) com crase (`).

1. Aspas Simples ()

As **aspas simples** são a forma mais tradicional e simples de criar strings em JavaScript. Ao usar aspas simples, você define uma sequência de caracteres entre elas.

```
let saudacao = 'Olá, Mundo!';
console.log(saudacao); // Saída: Olá, Mundo!
```

Você pode usar **aspas simples** para criar strings, mas se a string contiver uma aspa simples dentro dela, será necessário escapá-la usando a barra invertida (\).

```
let frase = 'O gato de João\'s foi embora.';
console.log(frase); // Saída: O gato de João's foi embora.
```

2. Aspas Duplas ("")

As **aspas duplas** funcionam da mesma forma que as aspas simples, mas com a diferença de que você pode usar aspas duplas dentro da string sem a necessidade de escape.

```
let saudacao = "Olá, Mundo!";
console.log(saudacao); // Saída: Olá, Mundo!
```

Aspas duplas são uma boa opção quando a string contém aspas simples, como em frases que indicam posse ou citações.

```
let frase = "Ele disse: \"Olá, tudo bem?\"";
console.log(frase); // Saída: Ele disse: "Olá, tudo bem?"
```

3. Template Literals (ou Template Strings) com Crase (`)

Os **template literals**, também conhecidos como **template strings**, são uma das características mais poderosas do JavaScript moderno. Eles permitem que você crie strings de forma mais flexível, com a vantagem adicional de poderem **incluir expressões** dentro delas, o que facilita a interpolação de variáveis ou cálculos diretamente na string.

Para criar um template literal, você usa **crase** (`) ao invés de aspas simples ou duplas.

```
let nome = 'João';
let saudacao = `Olá, ${nome}!`;
console.log(saudacao); // Saída: Olá, João!
```

No exemplo acima, \${nome} é uma **interpolação** que insere o valor da variável nome dentro da string.

Benefícios dos Template Literals:

- **Interpolação de variáveis e expressões:** Você pode inserir variáveis ou até mesmo expressões complexas dentro das strings de maneira muito simples.

```
let x = 10;
let y = 5;
let resultado = `A soma de ${x} e ${y} é ${x + y}.`;
console.log(resultado); // Saída: A soma de 10 e 5 é 15.
```



Quebras de linha: Ao usar template literals, você pode criar strings que ocupam várias linhas sem a necessidade de concatenar ou adicionar caracteres de quebra de linha manualmente.

Exemplo com múltiplas linhas:

```
let texto = `Este é um exemplo  
de uma string que ocupa  
várias linhas.`;  
console.log(texto);  
// Saída:  
// Este é um exemplo  
// de uma string que ocupa  
// várias linhas.
```

Cada tipo de string em JavaScript tem suas aplicações e vantagens específicas. As **aspas simples** e **aspas duplas** são ótimas para strings simples e quando a escolha entre uma ou outra depende do estilo ou necessidade de escape de caracteres. Já os **template literals** são extremamente úteis para a criação de strings mais dinâmicas, com a capacidade de incluir variáveis e até expressões dentro delas de forma intuitiva e legível. Saber quando usar cada uma dessas opções vai ajudar a escrever código mais limpo e eficiente!

Comparação entre as três opções:

Tipo de Aspas	Como Criar	Vantagens	Quando Usar
Aspas Simples (')	'Texto aqui'	Simples, direto	Quando não precisar de interpolação e evitar aspas duplas dentro da string
Aspas Duplas (")	"Texto aqui"	Permite usar aspas simples dentro da string	Quando precisar usar aspas simples dentro da string sem escapar
Template Literals (Crase) (`)	`Texto aqui \${variáveis ou expressões}` ``	Interpolação de variáveis e expressões, múltiplas linhas	Quando precisar de interpolação ou uma string de várias linhas

No JavaScript, **concatenar strings** e acessar **caracteres individuais** dentro de uma string são operações muito comuns. Vamos explorar como podemos realizar essas tarefas utilizando **aspas simples** ('') ou **aspas duplas** ("") e também como acessar os caracteres de uma string utilizando o índice.

1. Concatenando Strings

Usando Aspas Simples ou Duplas

A **concatenação** de strings é o processo de unir duas ou mais strings em uma só. Você pode usar tanto **aspas simples** quanto **aspas duplas** para criar e concatenar strings em JavaScript. A diferença é apenas o estilo de citação utilizado — o comportamento é o mesmo.

```
let nome = 'Maria';
let saudacao = 'Olá, ' + nome + '!';
console.log(saudacao); // Saída: Olá, Maria!
```

```
let nome = "Carlos";
let saudacao = "Bem-vindo, " + nome + "!";
console.log(saudacao); // Saída: Bem-vindo, Carlos!
```

Observação:

- A operação de **concatenação** é feita com o operador +. Quando você usa esse operador entre strings, o JavaScript as une em uma única string.
- Você pode concatenar tanto strings criadas com aspas simples quanto com aspas duplas. A escolha entre aspas simples ou duplas é apenas uma questão de estilo ou conveniência, mas não afeta a operação de concatenação.

Acessando Caracteres Individuais de uma String

Strings em JavaScript são **sequências de caracteres**. Cada caractere em uma string está localizado em uma **posição específica** chamada **índice**. O índice das strings começa do número 0, ou seja, o primeiro caractere tem o índice 0, o segundo caractere tem o índice 1, e assim por diante.

Exemplo de Acesso a um Caractere Específico:

```
let palavra = 'JavaScript';
console.log(palavra[0]); // Saída: 'J'
console.log(palavra[4]); // Saída: 'S'
```

No exemplo acima, palavra[0] acessa o primeiro caractere da string ('J'), e palavra[4] acessa o quinto caractere ('S').

Observação Importante:

- **Índices negativos** não funcionam da mesma maneira em JavaScript. Ou seja, palavra[-1] não retornará o último caractere. Para acessar o último caractere de uma string, você pode usar o método .length junto com o índice correto.

```
let palavra = 'JavaScript';
console.log(palavra[palavra.length - 1]); // Saída: 't'
```

O código palavra.length - 1 calcula a posição do último caractere, pois palavra.length retorna o número total de caracteres na string.

Técnica	Exemplo	Descrição
Concatenando com <code>+</code>	<code>'Olá, ' + nome + '!'</code>	Usa o operador <code>+</code> para juntar as strings.
Concatenando com Template Literals	<code>`Olá, \${nome}!`</code>	Usa template literals para interpolar variáveis diretamente.
Acessando Caracteres	<code>palavra[0]</code>	Acessa um caractere específico de uma string pelo índice.

Concatenar strings em JavaScript pode ser feito de várias maneiras. A concatenação simples com o operador `+` é muito comum, mas a utilização de **template literals** oferece uma abordagem mais moderna, legível e poderosa, especialmente quando há necessidade de incluir variáveis dentro das strings.

Acessar caracteres individuais dentro de uma string é feito utilizando o índice da string, lembrando que o índice começa de 0. Isso permite acessar e manipular os caracteres de maneira flexível e eficiente.

Em JavaScript, o tipo **number** é utilizado para representar **valores numéricos**, incluindo tanto números inteiros quanto números de **ponto flutuante** (decimais). O tipo number é um tipo de dado fundamental na linguagem, utilizado em uma ampla gama de operações, desde cálculos simples até manipulações mais complexas de dados.

Tipo Number

O tipo number em JavaScript abrange **todos os tipos numéricos**, incluindo tanto **inteiros** (números sem casas decimais) quanto **números de ponto flutuante** (números com casas decimais). Não há uma distinção explícita entre esses dois tipos em JavaScript, já que todos os números, inteiros ou flutuantes, são tratados de maneira similar como **valores de ponto flutuante de precisão dupla** no padrão IEEE 754.

Exemplos de uso do tipo number:

```
let inteiro = 42;      // Um número inteiro
let decimal = 3.14;    // Um número de ponto flutuante
```

Ambos os valores são armazenados como **números de ponto flutuante** internamente, o que significa que o JavaScript não diferencia explicitamente entre inteiros e números decimais.

Ponto Flutuante

Em JavaScript, números de **ponto flutuante** (ou **decimais**) são números que contêm casas decimais. Eles são chamados de "ponto flutuante" porque a posição da vírgula (ou ponto) pode variar, ou "flutuar", dependendo do número, o que permite que números muito grandes ou muito pequenos sejam representados.

Por exemplo:

```
let pi = 3.14159;      // Um número de ponto flutuante
let negativo = -0.0005; // Outro número de ponto flutuante
```

Importante sobre ponto flutuante:

Os números de ponto flutuante em JavaScript são representados de acordo com o padrão **IEEE 754 de precisão dupla**, que oferece uma **precisão limitada** e pode levar a **problemas de arredondamento**.

Exemplo de Problema de Precisão:

```
let resultado = 0.1 + 0.2;
console.log(resultado); // Saída: 0.3000000000000004
```

Neste caso, o resultado esperado seria 0.3, mas devido à representação interna do ponto flutuante, o valor retorna com uma pequena imprecisão. Isso é um comportamento comum em linguagens que utilizam esse tipo de representação.

NaN (Not-a-Number)

O valor **NaN** (Not-a-Number) é um valor especial em JavaScript que representa **algo que não é um número**. Isso pode ocorrer quando você tenta realizar uma operação que não resulta em um número válido.

Exemplos que resultam em NaN:

- Divisão de zero por zero.
- Tentativa de converter uma string que não é um número em um número.

```
let resultado1 = 0 / 0;           // NaN
let resultado2 = Math.sqrt(-1); // NaN (não é possível calcular a raiz quadrada de um número negativo)
let numero = Number("texto");   // NaN (não é possível converter uma string não numérica em número)
```

Características do NaN:

- **Não é igual a nenhum outro valor**, nem mesmo a ele mesmo. Isso significa que `NaN === NaN` resulta em `false`.

Infinity e -Infinity

Infinity e **-Infinity** são valores especiais em JavaScript que representam **valores numéricos infinitos**. Esses valores surgem quando um cálculo excede os limites de números representáveis.

Infinity: Representa um valor positivo infinito.

```
let positivoInfinito = 1 / 0; // Divisão por zero resulta em Infinity  
console.log(positivoInfinito); // Saída: Infinity
```

-Infinity: Representa um valor negativo infinito.

```
let negativoInfinito = -1 / 0; // Divisão de um número negativo por zero  
console.log(negativoInfinito); // Saída: -Infinity
```

Características de Infinity e -Infinity:

- Ambos são considerados **números válidos** no JavaScript e podem ser usados em operações matemáticas.
- **Infinity** e **-Infinity** são maiores e menores, respectivamente, do que qualquer outro número em JavaScript.

Os **valores booleanos** são um dos tipos de dados fundamentais em muitas linguagens de programação, incluindo o JavaScript. Para entender como eles funcionam, podemos começar com um conceito básico de **sistema binário**, que é a base do funcionamento dos computadores e, por conseguinte, dos valores booleanos.

1. Sistema Binário e Valores Booleanos

O **sistema binário** é um sistema numérico que utiliza apenas **dois valores: 0 e 1**. Isso ocorre porque, internamente, os computadores funcionam com circuitos elétricos que podem estar em dois estados possíveis: **ligado** ou **desligado**. Esses dois estados podem ser representados pelos números **0** (desligado) e **1** (ligado).

Comparação: Sistema Binário e Booleano

- No sistema binário, temos apenas dois dígitos: **0** e **1**. Eles representam os estados de "desligado" e "ligado", respectivamente.
- De forma semelhante, em programação, os valores **booleanos** são utilizados para representar **dois estados possíveis: false** (falso) e **true** (verdadeiro).

Assim, o sistema binário de 0 e 1 se conecta diretamente ao conceito de booleanos, que também lidam com duas condições mutuamente exclusivas: **false** (falso) e **true** (verdadeiro).

- **0: Decimal 0 = Binário 0**
- **1: Decimal 1 = Binário 1**
- **2: Decimal 2 = Binário 10**
- **3: Decimal 3 = Binário 11**
- **4: Decimal 4 = Binário 100**



O Conceito de Booleano

Em **programação**, um valor **booleano** é um tipo de dado que pode ter apenas **dois valores possíveis**:

- **true** (verdadeiro): Representa uma condição verdadeira, ou algo que é afirmativo.
- **false** (falso): Representa uma condição falsa, ou algo que é negativo ou não verdadeiro.

Esses valores são muito utilizados em **condições** (como em if, while e outros comandos condicionais) para tomar decisões no código. Por exemplo, você pode ter uma condição que verifica se um usuário está logado em um site, e se essa condição for verdadeira (true), o sistema permitirá o acesso. Se a condição for falsa (false), o acesso será negado.

3. Por que Utilizamos Booleanos?

Os valores booleanos são fundamentais para **controle de fluxo** em programação. Eles ajudam o programa a tomar decisões com base em condições lógicas. Em JavaScript, por exemplo, você pode usar expressões booleanas para determinar o que deve acontecer em seguida no código.

```
let usuarioLogado = true;

if (usuarioLogado) {
    console.log("Acesso permitido.");
} else {
    console.log("Acesso negado.");
}
```

Neste exemplo, se a variável `usuarioLogado` for **true**, o sistema irá imprimir "Acesso permitido". Se for **false**, ele imprimirá "Acesso negado".



Booleanos e Comparações

Em JavaScript, muitos operadores de comparação resultam em valores booleanos. Por exemplo, o operador `==` verifica se dois valores são iguais e retorna **true** ou **false** dependendo do resultado:

```
let a = 10;
let b = 5;

console.log(a == b); // Saída: false
console.log(a != b); // Saída: true
```

Como converter um valor em um Booleano:

Você também pode **converter outros tipos de dados** em valores booleanos usando a função **Boolean()** ou a **dupla negação (!!)**. Isso pode ser útil para garantir que uma variável seja avaliada como true ou false em uma expressão condicional.

```
let valor1 = 1;           // Um número não zero
let valor2 = 0;           // O número zero

console.log(Boolean(valor1)); // Saída: true
console.log(Boolean(valor2)); // Saída: false

console.log (!!valor1); // Saída: true
console.log (!!valor2); // Saída: false
```



Conclusão: A Importância dos Booleanos

Os **valores booleanos** são essenciais para a lógica de programação. Eles nos permitem representar e manipular condições que têm apenas dois estados possíveis: verdadeiro ou falso. Através de comparações e expressões booleanas, podemos controlar o fluxo de execução de um programa, tomar decisões e fazer verificações essenciais.

Assim como no sistema binário, onde **0** e **1** representam os estados de **desligado** e **ligado**, os valores **false** e **true** em JavaScript nos ajudam a trabalhar com condições lógicas e a construir programas interativos e dinâmicos.



Em JavaScript, os tipos de dados **undefined** e **null** são usados para representar a ausência de valor ou a ausência de um objeto, mas eles são diferentes em sua aplicação e comportamento.

1. O Tipo **undefined**

O tipo **undefined** é um valor primitivo que significa "**não definido**". Ele é automaticamente atribuído a uma variável que foi declarada, mas ainda não foi inicializada com um valor. Ou seja, uma variável que existe, mas que ainda não tem um valor definido, tem o valor **undefined**.

Quando o undefined é utilizado:

- **Variáveis não inicializadas:** Se você declarar uma variável sem atribuir um valor a ela, seu valor será **undefined**.

```
let nome;  
console.log(nome); // Saída: undefined
```

Características do undefined:

- **Tipo undefined** é um tipo primitivo e tem seu próprio tipo.
- Pode ser **comparado** diretamente com undefined ou com outros valores, mas é importante entender que ele é **falsy**, ou seja, é avaliado como false em expressões condicionais.

2. O Tipo null

O tipo **null** é um valor primitivo que representa **ausência intencional de valor**. Diferente de undefined, que ocorre automaticamente quando uma variável não foi atribuída, **null** é explicitamente atribuído a uma variável para indicar que ela não possui um valor ou que um valor foi **intencionalmente ausente**.

Quando o null é utilizado:

- **Atribuição explícita:** Você pode usar **null** quando deseja inicializar uma variável com um valor que representa ausência, mas de forma **intencional**.

```
let carro = null; // Nenhum valor para a variável, mas é intencional
console.log(carro); // Saída: null
```

Características do null:

- **Tipo null** é também um tipo primitivo, mas é considerado um **objeto** em JavaScript (o que é considerado um erro histórico da linguagem).
- **null** é um valor **falsy**, ou seja, quando avaliado em um contexto booleano, é tratado como false.

Característica	<code>undefined</code>	<code>null</code>
Significado	A variável foi declarada, mas não tem valor atribuído.	A variável foi intencionalmente definida como "não possui valor".
Atribuição automática	Acontece automaticamente quando uma variável é declarada sem valor.	Precisa ser atribuído explicitamente ao declarar a variável.
Tipo	Tipo primitivo <code>undefined</code> .	Tipo primitivo <code>object</code> (isso é um erro histórico em JavaScript).
Uso comum	Usado para variáveis não inicializadas e retorno de funções sem valor.	Usado para indicar a ausência de valor ou de objeto de forma intencional.

Comparações entre `undefined` e `null`

Quando comparados, `undefined` e `null` têm comportamentos específicos:

- **Compara `==`:** Quando utilizamos o operador de igualdade `==` (igualdade não estrita), o JavaScript considera `undefined` e `null` como **iguais**. No entanto, se comparados com `===` (igualdade estrita), eles são diferentes, já que têm tipos diferentes.

```
console.log(null == undefined); // Saída: true (são considerados iguais com ==)
console.log(null === undefined); // Saída: false (são de tipos diferentes)
```

Compara com outros valores: Ambos são considerados **falsy** em JavaScript. Ou seja, eles serão avaliados como false em uma expressão condicional.

```
if (null) {
  console.log("null é verdadeiro");
} else {
  console.log("null é falso!"); // Saída: null é falso!
}

if (undefined) {
  console.log("undefined é verdadeiro");
} else {
  console.log("undefined é falso!"); // Saída: undefined é falso!
}
```



- **undefined**: Significa que uma variável foi declarada, mas não recebeu valor. Ele é automaticamente atribuído pelo JavaScript.
- **null**: Significa que uma variável foi explicitamente definida para não ter valor. É um valor de ausência intencional.

Entender a diferença entre esses dois tipos de dado ajuda a evitar confusões e a escrever código mais preciso, especialmente em situações onde você lida com valores ausentes ou desconhecidos.

A **propriedade typeof** em JavaScript é um **operador** utilizado para determinar o tipo de dado de uma variável ou expressão. Ele retorna uma **string** que indica o tipo do valor que a variável ou expressão representa.

1. Sintaxe do typeof

A sintaxe do operador typeof é simples. Você apenas precisa escrever **typeof** seguido de uma variável ou expressão que você deseja verificar o tipo:

```
typeof expressão
```

O typeof pode retornar diferentes valores dependendo do tipo de dado da variável ou expressão. Aqui estão os tipos mais comuns que typeof pode retornar:

- **"undefined"**: Se a variável não foi definida ou se o valor dela é undefined.
- **null**: Como mencionado anteriormente, null é considerado um **objeto** quando utilizado com typeof, devido a um erro histórico da linguagem. Isso pode causar confusão, então, para verificar se algo é null, você pode usar uma comparação explícita.

```
let valor = null;  
console.log(typeof valor); // Saída: "object"  
console.log(valor === null); // Saída: true
```



Em JavaScript, **operadores aritméticos** são usados para realizar operações matemáticas básicas, como soma, subtração, multiplicação, divisão, entre outras. Eles são fundamentais para manipular números em programas e são amplamente utilizados em cálculos, processamento de dados, e nas mais diversas lógicas dentro de um código.

O que são Operadores Aritméticos?

Operadores aritméticos são símbolos que permitem realizar cálculos e operações matemáticas com variáveis ou valores numéricos. Eles trabalham diretamente com os **tipos numéricos** (inteiros e ponto flutuante), mas podem ser usados com outros tipos de dados, como strings, com comportamentos específicos.

Principais Operadores Aritméticos

Vamos explorar os operadores aritméticos mais comuns em JavaScript:

Adição (+)

O operador de adição é usado para somar dois valores.

- **Sintaxe:** `a + b`

```
let a = 10;
let b = 5;
let resultado = a + b;
console.log(resultado); // Saída: 15
```

Além de ser usado para somar números, o operador **+** também pode ser usado para concatenar **strings**. Se você usar o **+** entre uma string e um número, o JavaScript converterá automaticamente o número para string.

```
let texto = "Olá, o número é ";
let numero = 7;
let mensagem = texto + numero;
console.log(mensagem); // Saída: "Olá, o número é 7"
```



Subtração (-)

O operador de subtração é utilizado para subtrair um valor de outro.

- **Sintaxe:** $a - b$

```
let a = 10;
let b = 5;
let resultado = a - b;
console.log(resultado); // Saída: 5
```

Multiplicação (*)

O operador de multiplicação serve para multiplicar dois valores.

- **Sintaxe:** $a * b$

```
let a = 4;
let b = 3;
let resultado = a * b;
console.log(resultado); // Saída: 12
```



Divisão (/)

O operador de divisão é utilizado para dividir um valor por outro. Ele retorna o quociente da divisão.

- **Sintaxe:** a / b

```
let a = 10;
let b = 2;
let resultado = a / b;
console.log(resultado); // Saída: 5
```

Se a divisão não for exata, o resultado será um número decimal (ponto flutuante):

```
let a = 10;
let b = 3;
let resultado = a / b;
console.log(resultado); // Saída: 3.333333333333335
```

Módulo (%)

O operador de módulo retorna o **resto da divisão** entre dois números. Ele é útil para verificar se um número é divisível por outro ou para realizar operações que dependem de um valor restante após a divisão.

- **Sintaxe:** a % b

```
let a = 10;
let b = 3;
let resultado = a % b;
console.log(resultado); // Saída: 1
```



No exemplo anterior, 10 dividido por 3 tem um quociente de 3 e um resto de 1. O operador % retorna esse resto.

Outro uso comum do operador de módulo é para verificar se um número é par ou ímpar. Se o resto da divisão de um número por 2 for 0, o número é par; caso contrário, é ímpar:

```
let numero = 7;
if (numero % 2 === 0) {
    console.log("O número é par.");
} else {
    console.log("O número é ímpar.");
}
```

Operador	Descrição	Exemplo de uso
+	Adição (soma)	10 + 5 → 15
-	Subtração	10 - 5 → 5
*	Multiplicação	4 * 3 → 12
/	Divisão	10 / 2 → 5
%	Módulo (resto da divisão)	10 % 3 → 1

Além dos operadores aritméticos básicos (soma, subtração, multiplicação, etc.), JavaScript oferece alguns operadores aritméticos avançados que são extremamente úteis em cálculos mais complexos e em manipulações de variáveis. Vamos aprender sobre a **exponenciação** e os **operadores de incremento e decremento**.

1. Exponenciação (**)

O operador de **exponenciação** é utilizado para calcular uma base elevada a um certo expoente. Em outras palavras, ele permite calcular potências de números.

base ** expoente

```
let base = 2;  
let expoente = 3;  
let resultado = base ** expoente;  
console.log(resultado); // Saída: 8 (2 elevado à potência 3)
```

Incremento (++) e Decremento (--)

Os operadores **incremento** e **decremento** são usados para aumentar ou diminuir o valor de uma variável em **1** de forma rápida. Esses operadores podem ser usados tanto em sua forma **pós-fixa** quanto **pré-fixa**.

Operador de Incremento (++)

O operador de incremento **++** aumenta o valor de uma variável em **1**.

- **Pós-fixo** (a++): O valor da variável é aumentado depois que a expressão é avaliada.
- **Pré-fixo** (++a): O valor da variável é aumentado antes que a expressão seja avaliada.

Exemplo com pós-fixo:

```
let a = 5;  
console.log(a++); // Saída: 5 (Primeiro retorna 5, depois incrementa)  
console.log(a);   // Saída: 6 (Valor de 'a' foi incrementado para 6)
```

Exemplo com pré-fixo:

```
let a = 5;  
console.log(++a); // Saída: 6 (Primeiro incrementa, depois retorna 6)  
console.log(a);   // Saída: 6
```

Operador de Decremento (--)

O operador de decremento -- diminui o valor de uma variável em 1.

- **Pós-fixo** (a--): O valor da variável é diminuído depois que a expressão é avaliada.
- **Pré-fixo** (--a): O valor da variável é diminuído antes que a expressão seja avaliada.

Exemplo com pós-fixo:

```
let a = 5;
console.log(a--); // Saída: 5 (Primeiro retorna 5, depois decrementa)
console.log(a);   // Saída: 4 (Valor de 'a' foi decrementado para 4)
```

Exemplo com pré-fixo:

```
let a = 5;
console.log(--a); // Saída: 4 (Primeiro decrementa, depois retorna 4)
console.log(a);   // Saída: 4
```

Diferença entre Pré e Pós-Incremento/Decremento

A principal diferença entre o **pré-incremento** e o **pós-incremento** está no **momento em que o valor é alterado** e no **valor que é retornado**:

- **Pós-incremento (a++)**: O valor da variável é **usado** na expressão antes de ser incrementado (ou decrementado).
- **Pré-incremento (++a)**: O valor da variável é **alterado** antes de ser usado na expressão.

- O **operador de exponenciação (**)** facilita o cálculo de potências e raízes em JavaScript de maneira concisa e legível.
- Os **operadores de incremento (++)** e **decremento (--)** são extremamente úteis para ajustar o valor de uma variável rapidamente, sendo amplamente utilizados em loops e em manipulação de contadores.
- Entender as diferenças entre **pré** e **pós** incremento/decremento é essencial para evitar erros em expressões e garantir o funcionamento correto do seu código.

Operador	Descrição	Exemplo de uso
**	Exponenciação (base elevada ao expoente)	2 ** 3 → 8
++	Incremento (aumenta 1 no valor da variável)	a++ OU ++a
--	Decremento (diminui 1 no valor da variável)	a-- OU --a

Em JavaScript, os **operadores de comparação** são usados para comparar valores e retornar um valor **booleano** (verdadeiro ou falso). Eles são essenciais para realizar verificações e tomar decisões em seu código, como em estruturas condicionais (if, else, etc.). Vamos explorar cada um dos operadores de comparação e como utilizá-los.

1. O que são Operadores de Comparaçāo?

Os **operadores de comparação** são símbolos que comparam dois valores, verificando se uma condição é verdadeira ou falsa. Eles são frequentemente usados para controle de fluxo (como loops e condicionais), onde é necessário verificar se certos critérios são atendidos.

Esses operadores sempre retornam um valor **booleano**: true ou false.

2. Principais Operadores de Comparaçāo

Abaixo estão os operadores de comparação mais comuns em JavaScript:

Igualdade (==)

O operador **==** compara dois valores **somente em termos de valor**, ignorando o tipo dos dados. Ou seja, ele realiza uma comparação **não estrita**, o que pode causar alguns comportamentos inesperados.

```
console.log(5 == 5);      // Saída: true
console.log(5 == "5");    // Saída: true (a string "5" é convertida para o número 5)
console.log("10" == 10);  // Saída: true (a string "10" é convertida para o número 10)
```

Igualdade Estrita (==)

O operador `==` compara tanto o **valor** quanto o **tipo** dos operandos. Ou seja, para que a comparação seja verdadeira, os dois valores precisam ser **iguais em tipo e valor**.

```
console.log(5 === 5);    // Saída: true
console.log(5 === "5");  // Saída: false (tipos diferentes: número e string)
console.log("10" === 10); // Saída: false (tipos diferentes: string e número)
```

Desigualdade (!=)

O operador `!=` verifica se dois valores **não são iguais**, comparando apenas os **valores** e ignorando o tipo de dados.

```
console.log(5 != 10);    // Saída: true
console.log(5 != "5");   // Saída: false (pois o valor é o mesmo, embora os tipos sejam diferentes)
console.log("10" != 10); // Saída: false (mesmo valor, tipos diferentes)
```

Desigualdade Estrita (!==)

O operador **!==** verifica se dois valores **não são iguais** e **não são do mesmo tipo**. Ele compara **tanto o valor quanto o tipo**.

```
console.log(5 !== 10);    // Saída: true
console.log(5 !== "5");   // Saída: true (tipos diferentes: número e string)
console.log("10" !== 10); // Saída: true (tipos diferentes: string e número)
```

Maior que (>)

O operador **>** verifica se o valor à esquerda é **maior** que o valor à direita.

```
console.log(10 > 5);    // Saída: true
console.log(5 > 10);    // Saída: false
console.log(5 > 5);    // Saída: false
```

Maior ou igual a (>=)

O operador **>=** verifica se o valor à esquerda é **maior ou igual** ao valor à direita.

```
console.log(10 >= 5);    // Saída: true
console.log(5 >= 10);    // Saída: false
console.log(5 >= 5);    // Saída: true
```



Menor que (<)

O operador < verifica se o valor à esquerda é **menor** que o valor à direita.

```
console.log(5 < 10); // Saída: true
console.log(10 < 5); // Saída: false
console.log(5 < 5); // Saída: false
```

Menor ou igual a (<=)

O operador <= verifica se o valor à esquerda é **menor ou igual** ao valor à direita.

```
console.log(5 <= 10); // Saída: true
console.log(10 <= 5); // Saída: false
console.log(5 <= 5); // Saída: true
```

Por que usar a Comparação Estrita (==) em vez de (==)?

A principal diferença entre == e === é que == realiza uma comparação **não estrita**, tentando converter os valores para o mesmo tipo antes de compará-los. Isso pode levar a resultados inesperados quando você trabalha com tipos diferentes. Por exemplo, a comparação entre null e undefined com == retornará true, mas com === retornará false porque são tipos diferentes.

```
console.log(null == undefined); // Saída: true
console.log(null === undefined); // Saída: false
```



Operadores de Comparação e Tipos Não Numéricos

Os operadores de comparação também podem ser usados com **strings** e **booleans**. Ao comparar strings, a comparação é feita de acordo com a ordem lexicográfica (semelhante ao dicionário). Com booleanos, true é considerado maior que false.

```
console.log("apple" > "banana"); // Saída: false (pois "apple" vem antes de "banana")
console.log("apple" < "banana"); // Saída: true
console.log("apple" === "apple"); // Saída: true
```

```
console.log(true > false); // Saída: true (true é maior que false)
console.log(true === true); // Saída: true
console.log(false === true); // Saída: false
```

Operador	Descrição	Exemplo
<code>==</code>	Igualdade (não estrita)	<code>5 == "5"</code> → true
<code>===</code>	Igualdade estrita (valor e tipo)	<code>5 === "5"</code> → false
<code>!=</code>	Desigualdade (não estrita)	<code>5 != "5"</code> → false
<code>!==</code>	Desigualdade estrita (valor e tipo)	<code>5 !== "5"</code> → true
<code>></code>	Maior que	<code>10 > 5</code> → true
<code>>=</code>	Maior ou igual a	<code>10 >= 5</code> → true
<code><</code>	Menor que	<code>5 < 10</code> → true
<code><=</code>	Menor ou igual a	<code>5 <= 10</code> → true



A **coerção implícita** é o processo no qual o JavaScript converte automaticamente os tipos de dados de uma expressão para tentar realizar uma operação. Isso ocorre em situações onde os operadores esperam tipos específicos, como números ou strings, mas recebem valores de tipos diferentes.

Embora a coerção implícita possa ser conveniente, ela também pode gerar resultados inesperados, então é importante entender como o JavaScript lida com esses casos.

```
console.log(5 === "5");
console.log(5 + "5"); // número é convertido para string
console.log("10" - 5); // string é convertida para number
console.log("3" * "2"); // strings são convertidas para number
```

Exemplo 1: 5 + "5"

Neste caso, temos um número (5) e uma string ("5"). O operador **+** em JavaScript tem dois propósitos: somar números ou concatenar strings. Quando pelo menos um dos operandos é uma string, o JavaScript converte o outro operando para string e realiza a concatenação.

- 5 (número) é convertido para "5" (string).
- As duas strings são concatenadas: "5" + "5" = "55".

Exemplo 2: "10" - 5

Aqui, temos uma string ("10") e um número (5). O operador `-` é estritamente numérico, ou seja, ele espera operar com números. Quando o JavaScript detecta que um dos operandos é uma string que pode ser convertida para número, ele realiza a conversão implícita (coerção) para que a operação seja possível.

- "10" (string) é convertido para 10 (número).
- A operação matemática é realizada: $10 - 5 = 5$.

Exemplo 3: "3" * "2"

Neste caso, temos duas strings ("3" e "2") e o operador `*`, que também é estritamente numérico. O JavaScript converte as strings para números automaticamente, pois a multiplicação não faz sentido com strings.

- "3" (string) é convertido para 3 (número).
- "2" (string) é convertido para 2 (número).
- A operação é realizada: $3 * 2 = 6$.

Por que isso acontece?

A coerção implícita ocorre porque o JavaScript tenta ser uma linguagem "flexível". Quando diferentes tipos de dados são usados em uma operação, o motor da linguagem converte um ou ambos os operandos para um tipo compatível com a operação.

Essa conversão é baseada em **regras internas de coerção**:

- Para o operador `+`:
 - Se um dos operandos for string, o outro é convertido para string e ocorre a **concatenação**.
- Para operadores como `-`, `*`, `/`:
 - Ambos os operandos são convertidos para número, sempre que possível.
- Se a coerção falhar (por exemplo, quando a string não pode ser convertida para número), o resultado será **NaN** (Not a Number).

== (Igualdade Não Estrita)

O operador **==** verifica se dois valores são **iguais**, mas permite a conversão de tipos de dados antes da comparação. Isso significa que, se os dois valores tiverem tipos diferentes, o JavaScript tenta **converter um ou ambos os valores** para o mesmo tipo (coerção implícita) e, só depois, realiza a comparação.

Explicação:

- **Tipos diferentes:** "5" é uma string e 5 é um número.
- O JavaScript converte a string "5" para o número 5 (coerção implícita).
- Após a conversão, a comparação se torna: 5 == 5.
- Como os valores são iguais, o resultado é **true**.

```
console.log("5" == 5); // Saída: true
```

O que são null e undefined?

- **null**:
 - Representa a ausência intencional de qualquer valor.
 - Geralmente é atribuído manualmente para indicar que uma variável "não tem valor".
- **undefined**:
 - Indica que uma variável foi declarada, mas ainda não recebeu nenhum valor.
 - Também é retornado por funções que não possuem um valor explícito de retorno.

```
console.log(null == undefined);
```

Por que null == undefined é true?

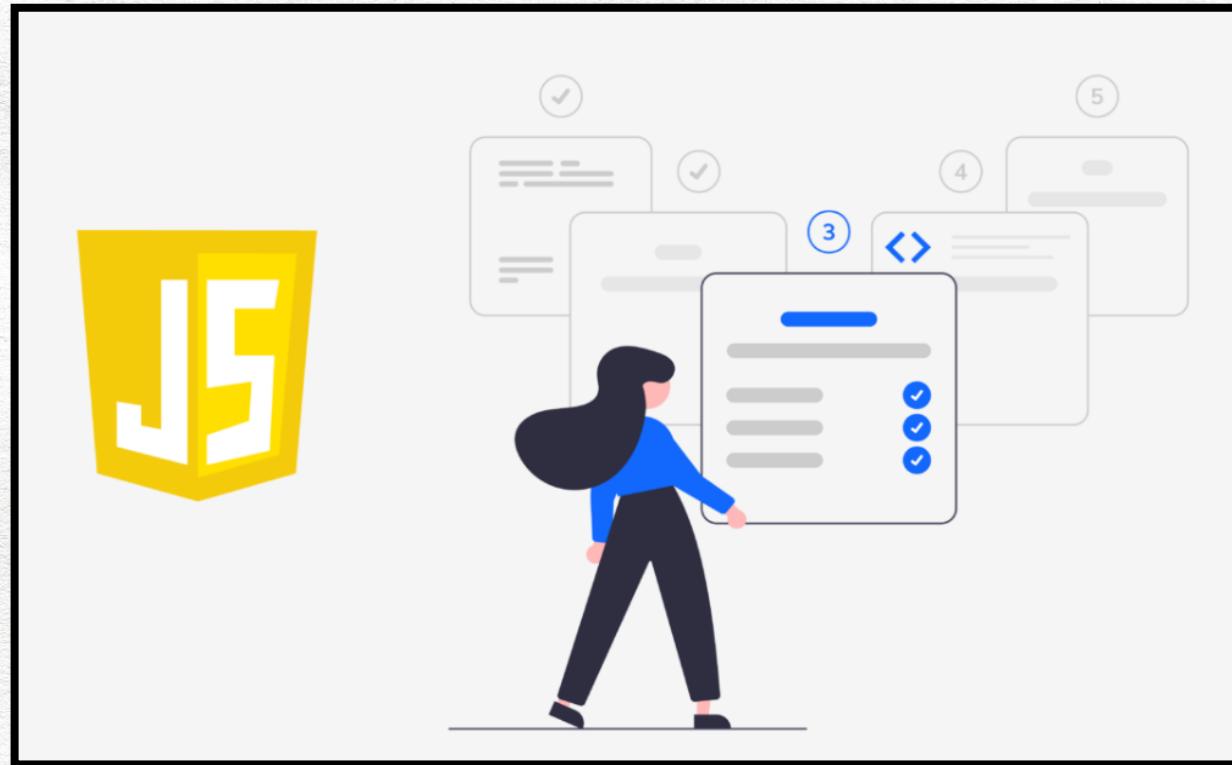
No JavaScript, o operador **==** realiza **coerção implícita** para determinar se dois valores são equivalentes. Segundo as regras do JavaScript definidas pela linguagem:

- **null e undefined são tratados como valores iguais** quando comparados com o operador de igualdade não estrita (==).
- Isso é porque ambos representam a ausência de valor de formas ligeiramente diferentes.



Conclusão

A coerção implícita é uma funcionalidade poderosa do JavaScript que pode simplificar operações, mas também é uma fonte de bugs. Entender como ela funciona nos diferentes operadores permite que você escreva código mais previsível e evite armadilhas. Ao trabalhar com operações matemáticas ou manipulação de strings, lembre-se das regras de coerção para antecipar os resultados!



Os métodos de **conversão explícita** no Javascript são ferramentas ou funções que permitem ao desenvolvedor converter um valor de um tipo de dado para outro de forma **intencional** e **direta**, sem depender de coerções implícitas realizadas automaticamente pela linguagem.

Essas conversões são utilizadas para garantir que os valores tenham o tipo correto antes de serem usados em operações ou comparações, evitando resultados inesperados. Os métodos mais comuns incluem conversões para números, strings ou booleanos. A principal característica da conversão explícita é a **clareza** e o **controle** que ela proporciona no código, tornando-o mais previsível e fácil de entender.

TYPEOF

A propriedade **typeof** no JavaScript é um operador utilizado para verificar o **tipo de dado** de um valor ou variável. Ele retorna uma **string** que descreve o tipo do valor avaliado.

Finalidade

A principal utilidade do **typeof** é ajudar os desenvolvedores a identificar o tipo de dado de uma variável ou expressão durante o desenvolvimento, depuração ou validação de entradas.

```
let numero = 123;  
console.log(typeof numero);
```

A **conversão explícita** no JavaScript ocorre quando usamos métodos ou funções para transformar manualmente um valor de um tipo de dado para outro. Isso oferece maior controle sobre como os dados são manipulados e evita os comportamentos inesperados que podem ocorrer com a coerção implícita.

Aqui estão os principais métodos para realizar conversões explícitas:

1. Number()

O método **Number()** converte o valor fornecido para o tipo number (número).

- **Valores válidos:** Strings numéricas, booleanos (true e false), e alguns outros tipos podem ser convertidos para números.
- **Valores inválidos:** Strings não numéricas ou valores que não podem ser interpretados como números resultam em **NaN** (Not-a-Number).

2. String()

O método **String()** converte qualquer valor para o tipo string.

- **Conversão universal:** Funciona para números, booleanos, objetos, arrays, e até mesmo valores como null e undefined.
- **Uso:** Útil para transformar valores em representações textuais, especialmente quando precisamos concatenar ou exibir informações.

3. Boolean()

O método **Boolean()** converte valores para o tipo boolean (true ou false).

- **Valores que se tornam false (falsy):** 0, "" (string vazia), null, undefined, NaN.
- **Todos os outros valores:** São considerados **true** (truthy).

4. `.toString()`

O método `.toString()` é usado em objetos, números e outros tipos para convertê-los em strings. Ele é um método de instância, ou seja, deve ser chamado em um valor ou variável.

- **Diferença em relação a `String()`:** Enquanto `String()` pode ser usado diretamente em qualquer valor, `.toString()` só pode ser chamado em tipos específicos (como números, arrays, ou objetos). Ele pode gerar um erro se for chamado em `null` ou `undefined`.

Comparação Entre `String()` e `.toString()`

Método	Aplicação	Limitação
<code>String()</code>	Conversão genérica e direta.	Funciona com qualquer valor.
<code>.toString()</code>	Método específico de instância.	Pode gerar erro em <code>null</code> ou <code>undefined</code> .

Os operadores lógicos no JavaScript são usados para realizar operações booleanas, ou seja, eles trabalham com valores do tipo **true** e **false**. Eles são amplamente utilizados em condições, expressões de controle de fluxo e validações.

Principais Operadores Lógicos

- **&& (AND - E Lógico)**

- Retorna **true** se **ambos os operandos** forem verdadeiros.
- Caso contrário, retorna **false**.
- Ele avalia os valores da esquerda para a direita e retorna o primeiro valor **falsy** ou o último valor, se todos forem **truthy**.

Operando 1	Operando 2	Resultado
true	true	true
true	false	false
false	true	false
false	false	false

• || (OR - OU Lógico)

- Retorna **true** se **pelo menos um dos operandos** for verdadeiro.
- Caso todos os operandos sejam falsos, retorna **false**.
- Avalia da esquerda para a direita e retorna o primeiro valor **truthy** ou o último valor, se todos forem **falsy**.

OR ()		
Operando 1	Operando 2	Resultado
true	true	true
true	false	true
false	true	true
false	false	false

- **! (NOT - Negação Lógica)**

- Inverte o valor lógico.
- Se o operando for **true**, ele retorna **false**, e vice-versa.

NOT (!)	
Operando	Resultado
true	false
false	true

```
let a = true;
let b = false;

console.log(a && b); // false (E Lógico: ambos precisam ser verdadeiros)
console.log(a || b); // true  (OU Lógico: pelo menos um é verdadeiro)
console.log(!a);    // false (Negação: inverte o valor de `a`)
```

Combinação de Operadores

Os operadores lógicos podem ser combinados para construir expressões mais complexas, e a **ordem de precedência** define como eles são avaliados:

- **!** (NOT) tem a maior precedência.
- **&&** (AND) vem em seguida.
- **||** (OR) tem a menor precedência.

Os operadores lógicos são ferramentas poderosas para controlar o fluxo lógico de um programa, permitindo validações eficientes e expressões condicionais robustas. Compreender seu comportamento e como combiná-los é essencial para criar código lógico e funcional.

```
let x = true;
let y = false;

console.log(!x || (x && y));
// Avaliação:
// 1. !x -> false
// 2. x && y -> false
// 3. false || false -> false
```



Operações booleanas envolvem trabalhar com valores **lógicos: true e false**. Essas operações são a base da lógica computacional e são amplamente utilizadas para controlar o fluxo de execução em programas.

O que são Operações Booleanas?

Uma **operação booleana** é qualquer expressão que resulta em um valor lógico (**true** ou **false**). Essas operações geralmente envolvem operadores lógicos, operadores de comparação e valores booleanos diretamente.

Categorias de Operações Booleanas

- **Operadores Lógicos**

Usados para combinar ou modificar valores booleanos:

- **&& (AND - E lógico):** Retorna true se ambos os operandos forem true.
- **|| (OR - OU lógico):** Retorna true se pelo menos um dos operandos for true.
- **! (NOT - Negação lógica):** Inverte o valor lógico de um operando.

- **Operadores de Comparação**

Avaliam a relação entre dois valores:

- **==:** Verifica igualdade com coerção de tipo.
- **===:** Verifica igualdade sem coerção de tipo.
- **!= e !=:** Verificam desigualdade.
- **>, <, >=, <=:** Verificam relações de ordem (maior, menor, etc.).

Por que Operações Booleanas são Importantes?

- Controle de Fluxo:
 - Usadas em estruturas como if, else, while, e for para determinar quais blocos de código devem ser executados.
- Validação de Dados:
 - Garantem que os valores fornecidos são válidos antes de realizar operações.
- Lógica Condisional:
 - Permitem construir expressões dinâmicas e decisões complexas no código.

Exemplos

Operador AND (&&)

O operador **&&** retorna **true** se **ambas as condições** forem verdadeiras. Caso qualquer uma delas seja **falsa**, ele retorna **false**.

```
let podeFazerLogin = idade >= 18 && condigoPromocional; // true  
console.log(podeFazerLogin);
```

- **idade >= 18:** Verifica se a idade é maior ou igual a 18. Resultado: **true** (porque idade é 20).
- **condigoPromocional:** O valor já é **true**.
- **Resultado final:** Como ambas as condições são verdadeiras, o valor de podeFazerLogin será **true**.

```
let podeFazerLogin2 = idade2 >= 18 && condigPromocional; // false  
console.log(podeFazerLogin2);
```

- `idade2 >= 18`: Verifica se a idade é maior ou igual a 18. Resultado: **false** (porque `idade2` é 15).
- `condigPromocional`: É **true**, mas o operador `&&` exige que ambas as condições sejam verdadeiras.
- **Resultado final:** Como uma das condições é falsa, o valor de `podeFazerLogin2` será **false**.

Operador OR (||)

O operador `||` retorna **true** se pelo menos uma das condições for verdadeira. Ele avalia da esquerda para a direita e para no primeiro valor **truthy**.

```
let loginOr = idade >= 18 || condigPromocional; // true  
console.log(loginOr);
```

- `idade >= 18`: É **true** (porque `idade` é 20).
- O operador para aqui, já que encontrou uma condição verdadeira.
- **Resultado final:** `loginOr` será **true**.

```
let loginOr2 = idade2 >= 18 || condigopromocional; // true  
console.log(loginOr2);
```

- `idade2 >= 18`: É **false** (porque `idade2` é 15).
- `condigopromocional`: É **true**.
- **Resultado final:** Pelo menos uma condição é verdadeira, então `loginOr2` será **true**.

```
let promocional = false;  
console.log(idade2 >= 18 || promocional); // false
```

- `idade2 >= 18`: É **false**.
- `promocional`: Também é **false**.
- **Resultado final:** Ambas as condições são falsas, então o valor retornado será **false**.

Operador NOT (!)

O operador **!** inverte o valor booleano do operando. Se o valor é **true**, ele se torna **false**, e vice-versa.

```
let perfilConfigurado = false;  
let alerta = !perfilConfigurado;  
console.log(alerta); // true
```

- **perfilConfigurado:** É **false**.
- **!perfilConfigurado:** O operador **!** inverte o valor, tornando-o **true**.
- **Resultado final:** alerta será **true**, indicando que o perfil **não está configurado**.

O operador lógico **!** (NOT) é utilizado para **inverter** o valor lógico de uma expressão ou variável. Ele é um dos operadores mais simples e ao mesmo tempo poderoso em JavaScript, permitindo manipular valores booleanos de forma direta.

Como o **!** Funciona?

O operador **!** trabalha da seguinte forma:

- **Se o valor for true**, ele retorna **false**.
- **Se o valor for false**, ele retorna **true**.

Este comportamento faz dele uma ferramenta fundamental para **negar condições** ou **verificar valores falsy/truthy**.



```

let ativo = true;
console.log(!ativo); // false (inverteu o valor de ativo)

let desativado = false;
console.log(!desativado); // true (inverteu o valor de desativado)

```

No exemplo acima:

- O valor de ativo era **true**, e ao aplicar !ativo, o resultado se tornou **false**.
- O valor de desativado era **false**, e ao aplicar !desativado, o resultado se tornou **true**.

Combinação com Valores Não-Booleanos

Em JavaScript, o operador **!** também converte implicitamente valores não-booleanos em booleanos, antes de inverter o resultado. Essa conversão segue as regras de **truthy** e **falsy**:

Valores Falsy (avaliados como false)

- false
- 0
- "" (string vazia)
- null
- undefined
- NaN

Valores Truthy (avaliados como true)

- Qualquer valor que **não seja falsy**, como:
 - Strings não vazias ("hello")
 - Números diferentes de zero (42, -1)
 - Arrays ([] e objetos {})

```

console.log(!0);           // true (0 é falsy, invertido para true)
console.log!("");          // true (string vazia é falsy)
console.log(!42);          // false (42 é truthy)
console.log!("Hello");     // false (string não vazia é truthy)
console.log(!null);        // true (null é falsy)
console.log(!undefined);   // true (undefined é falsy)

```



Comparação entre Tipos Primitivos e Tipos por Referência

No JavaScript, os **tipos primitivos** e os **tipos por referência** têm comportamentos distintos devido à forma como armazenam e manipulam dados. Essa diferença é essencial para entender como variáveis funcionam em diferentes contextos.

1. Definição

- **Tipos Primitivos:**

Armazenam o **valor diretamente** na variável. Cada variável tem sua própria cópia independente do valor.

- **Tipos por Referência:**

Armazenam uma **referência** ao local da memória onde os dados reais estão localizados. Múltiplas variáveis podem compartilhar a mesma referência, apontando para o mesmo dado.

2. Mutabilidade

- **Tipos Primitivos** são **imutáveis**:

O valor em si não pode ser alterado. Se uma operação parece alterar o valor, ela retorna um novo valor.

- **Tipos por Referência** são **mutáveis**:

O dado pode ser alterado diretamente sem mudar a referência.

Tipos de Dados por Referência no JavaScript

Em JavaScript, os **tipos de dados por referência** (ou **referenciais**) são aqueles em que as variáveis não armazenam diretamente o valor do dado, mas sim uma **referência** ao local da memória onde o dado está armazenado. Isso é diferente dos **tipos primitivos**, que armazenam diretamente os valores.

Características dos Tipos de Dados por Referência

- **Armazenamento por Referência**
 - Ao criar uma variável com um tipo de dado referencial, ela não contém o valor em si, mas sim uma "ponte" para o local onde o dado está armazenado na memória.
 - Se você copiar uma variável que contém um tipo referencial, ambas as variáveis apontarão para o mesmo dado na memória.
- **Mutabilidade**
 - Os tipos de dados por referência são **mutáveis**, ou seja, você pode alterar o conteúdo do dado sem alterar a referência.
 - Quando você modifica o dado usando uma das variáveis, todas as variáveis que apontam para essa referência veem a mudança.
- **Comparação por Referência**
 - Quando você compara dois tipos referenciais usando `==` ou `===`, o JavaScript verifica se ambas as variáveis apontam para o mesmo local na memória, e **não os valores internos**.

Principais Tipos de Dados por Referência

Objetos (Object)

- Estruturas de dados que armazenam pares de chave-valor.
- Usados para modelar dados mais complexos.

Arrays (Array)

- Coleções ordenadas de dados.
- Embora sejam objetos, eles têm funcionalidades específicas para lidar com listas.

Funções (Function)

- Em JavaScript, as funções também são tipos referenciais.
- Elas podem ser armazenadas em variáveis, passadas como argumentos e retornadas de outras funções.



Diferença entre Tipos Primitivos e Referenciais

Característica	Tipos Primitivos	Tipos Referenciais
Armazenamento	Valor diretamente	Referência ao local na memória
Mutabilidade	Imutável	Mutável
Comparação	Compara valores	Compara referências
Cópia de Variável	Cria uma nova cópia	Compartilha a mesma referência
Exemplos	<code>number</code> , <code>string</code> , <code>boolean</code>	<code>object</code> , <code>array</code> , <code>function</code>

Os tipos de dados por referência são fundamentais para trabalhar com dados complexos em JavaScript, permitindo flexibilidade na manipulação e compartilhamento de informações dentro do programa.

Conclusão

- **Tipos Primitivos** são ideais para dados simples e imutáveis, como números e textos.
- **Tipos Referenciais** são usados para dados complexos e estruturados, permitindo maior flexibilidade e compartilhamento de informações.

Um **array** é um tipo de dado que permite armazenar múltiplos valores em uma única variável. Diferente de variáveis simples, que armazenam apenas um valor, os arrays podem armazenar **listas** de valores, podendo esses valores ser de qualquer tipo: números, strings, booleanos, ou até mesmo outros arrays e objetos.

- **Índice:** Cada valor dentro de um array é associado a um **índice**, que é uma posição numérica que começa em 0. Isso significa que o primeiro item de um array está no índice 0, o segundo no índice 1, e assim por diante.
- **Mutabilidade:** Arrays são **mutáveis**, ou seja, seus valores podem ser alterados após a criação do array. No JavaScript, os arrays são tratados como objetos, e isso permite que você altere os elementos de um array a qualquer momento.

Criação de um Array

```
let lista = ["Banana", 23, true, "Maçã"]; // índice 0
```

Aqui, estamos criando um array chamado lista. Esse array contém quatro elementos:

- No **índice 0**: "Banana" (uma string)
- No **índice 1**: 23 (um número)
- No **índice 2**: true (um valor booleano)
- No **índice 3**: "Maçã" (uma string)

O array é criado com diferentes tipos de dados, o que é totalmente válido em JavaScript, pois um array pode armazenar qualquer tipo de valor, misturando números, strings, e outros tipos.

Atribuindo um Novo Array

```
lista = ["banana", "maçã", "pera"];
console.log(lista);
```

Neste momento, o conteúdo do array lista foi substituído por um novo array com três elementos:

- "banana" (índice 0)
- "maçã" (índice 1)
- "pera" (índice 2)
- O console.log(lista) imprime o conteúdo do array atualizado: ["banana", "maçã", "pera"]

Modificando um Valor de um Índice

```
lista[0] = "laranja";
console.log(lista[0]);
console.log(lista);
```

- **lista[0] = "laranja";**: Aqui, estamos alterando o valor do índice 0 do array lista. Antes, o valor era "banana", e agora passamos a ser "laranja".
- **console.log(lista[0]);**: Imprime o novo valor armazenado no índice 0, que é "laranja".
- **console.log(lista);**: Imprime todo o array após a alteração, mostrando que o array agora é ["laranja", "maçã", "pera"].



Tentando Modificar um Caractere de uma String

```
let nome = "João";
nome[0] = "M";
console.log(nome);
```

- Aqui, temos uma string chamada nome com o valor "João".
- Em JavaScript, **strings são imutáveis**, ou seja, você não pode alterar um caráter específico de uma string diretamente. A tentativa de fazer nome[0] = "M"; não irá funcionar. Isso não altera o primeiro caractere da string.
- A console.log(nome) irá imprimir a string original, que é "João", sem alteração.

Destaques Importantes:

- **Arrays** são **mutáveis**. Você pode alterar seus valores diretamente, como mostramos com o exemplo de lista[0] = "laranja".
- **Strings** são **imutáveis**. Mesmo que você tente modificar um caractere diretamente, o valor da string não será alterado, como demonstrado com a variável nome.

Resumo

- **Array**: Estrutura que armazena múltiplos valores, podendo ser de tipos diferentes. Utiliza índices numéricos para acessar os elementos, começando do índice 0.
- **Mutabilidade**: Arrays podem ser modificados diretamente (como em lista[0] = "laranja"), mas strings não podem ser modificadas diretamente (como em nome[0] = "M").
- **Índices**: Usados para acessar e modificar os elementos de um array.

Criando um Array Simples e Acessando seus Elementos

```
let lista = ["Monitor", "Teclado", "Mouse"];

console.log(lista[0]); // "Monitor"
console.log(lista[1]); // "Teclado"
console.log(lista[2]); // "Mouse"
```

- **Array lista:** Criamos um array com três elementos: "Monitor", "Teclado" e "Mouse".
- **Índice de Acesso:** Para acessar os elementos de um array, utilizamos o **índice** do elemento. No JavaScript, os índices começam em 0. Ou seja:
 - lista[0] acessa o primeiro item (monitor),
 - lista[1] acessa o segundo item (teclado),
 - lista[2] acessa o terceiro item (mouse).

Esses valores são impressos no console.

Modificando um Elemento de um Array

```
lista[0] = "WebCam";
console.log(lista); // ["WebCam", "Teclado", "Mouse"]
```

- **Modificação de Elementos:** A sintaxe lista[0] = "WebCam" modifica o valor armazenado no **índice 0** do array lista. Antes, esse valor era "Monitor", e agora ele foi alterado para "WebCam".
- O comando console.log(lista) exibe o array após a modificação.

Acessando e Modificando um Índice Fora do Limite do Array

```
lista[4] = "Monitor";
console.log(lista); // ["WebCam", "Teclado", "Mouse", <1 empty item>, "Monitor"]
```

- **Adicionando um Elemento Fora do Limite:** No JavaScript, se você atribuir um valor a um índice que está fora do tamanho atual do array, o JavaScript cria "buracos" (elementos vazios) para os índices que estão entre o final do array e o índice fornecido.
- lista[4] = "Monitor"; adiciona o valor "Monitor" no índice 4, criando um "buraco" no índice 3, o que resulta no array ["WebCam", "Teclado", "Mouse", <1 empty item>, "Monitor"].

Propriedade length

```
console.log(lista.length); // 5
```

- **Propriedade length:** A propriedade length de um array retorna o número de elementos presentes no array. No exemplo acima, mesmo com o "buraco" criado no índice 3, o array possui 5 índices, portanto o valor de lista.length é 5.

Observação: O valor de length reflete o maior índice do array mais 1. Ou seja, se um array tem um índice 4, ele terá 5 elementos (contando do índice 0 ao 4).

```
lista.length = lista.length - 2; // 5  
console.log(lista);
```

Alterando o Tamanho do Array: Se você diminuir a propriedade length, o array será truncado. No código comentado, se fizermos lista.length = lista.length - 2, o array será cortado para ter apenas 3 elementos, pois estamos removendo 2 elementos.

Matrizes Bidimensionais (Arrays dentro de Arrays)

```
let matrizVendas = [
  [100, 200, 300],
  [400, 500, 50], //Loja B
  [700, 400, 450],
];

console.log(matrizVendas); // [[100, 200, 300], [400, 500, 50], [700, 400, 450]]
console.log(matrizVendas[1]); // [400, 500, 50]
```

Matriz Bidimensional: Uma matriz bidimensional é um **array de arrays**. Neste caso, matrizVendas é um array contendo três arrays internos:

- O primeiro array é [100, 200, 300], que representa os valores de vendas de uma loja.
- O segundo array é [400, 500, 50], representando a loja B.
- O terceiro array é [700, 400, 450].

Você pode acessar um array interno utilizando um índice, como matrizVendas[], que retorna o array [400, 500, 50] (vendas da loja B).

Modificando Elementos de Matrizes Bidimensionais

```
matrizVendas[1][2] = 500;  
console.log(matrizVendas[1]); // [400, 500, 500]
```

Alterando Elementos Internos: Dentro de uma matriz bidimensional, você pode modificar os elementos internos. No código acima, alteramos o valor da **posição [1][2]** da matrizVendas, ou seja, o terceiro elemento da segunda linha da matriz.

- O valor original era 50, e após a modificação, ele passa a ser 500.
- O console.log(matrizVendas[1]) mostra a linha atualizada: [400, 500, 500].

Calculando o Resultado de uma Loja

```
let resultadoLojaB = matrizVendas[1][0] + matrizVendas[1][1] + matrizVendas[1][2];  
console.log(resultadoLojaB); // 1400
```

Somando Elementos da Matriz: A soma dos valores das vendas da loja B (índice 1 da matriz) é feita somando os elementos matrizVendas[1][0], matrizVendas[1][1] e matrizVendas[1][2], ou seja, $400 + 500 + 500 = 1400$.

- O console.log(resultadoLojaB) imprime o resultado da soma, que é 1400.

Resumo dos Conceitos

- **Arrays:** Estrutura de dados que armazena múltiplos valores em uma única variável. Cada valor é acessado por um **índice** numérico, começando do **0**.
- **Modificação de Arrays:** Você pode alterar os valores de um array usando o índice, e o JavaScript permite a criação de "buracos" se você acessar um índice fora do limite do array.
- **Propriedade length:** Retorna o número total de elementos de um array, incluindo "buracos" criados.
- **Matrizes Bidimensionais:** São arrays dentro de arrays, permitindo armazenar dados de forma estruturada, como uma tabela.
- **Manipulação de Matrizes:** Você pode acessar e modificar elementos internos de uma matriz bidimensional com base nos dois índices.

Esse entendimento básico de **arrays** e **matrizes bidimensionais** é fundamental para organizar e manipular dados de forma eficiente em Javascript.

Assim como os **arrays**, os **objetos** também são tipos de dados compostos no Javascript, mas com uma estrutura diferente. Enquanto os arrays são indexados por números (índices), os objetos são **indexados por chaves** ou **nomes** (também chamados de propriedades). Essa estrutura de dados permite armazenar coleções de informações mais complexas, onde cada item é associado a uma chave que facilita a sua identificação e uso.

Criando um Objeto e Acessando suas Propriedades

```
let carro = {  
    marca: "Toyota",  
    modelo: "Corolla",  
    ano: 2024,  
    cor: "Prata",  
    airbag: true,  
    itens: ["abs", "4 portas", "step"],  
};
```

Objeto carro: Criamos um objeto chamado carro, que tem várias propriedades (ou **chaves**) associadas a valores:

- marca: uma string "Toyota",
- modelo: uma string "Corolla",
- ano: um número 2024,
- cor: uma string "Prata",
- airbag: um valor booleano true,
- itens: um array com os itens do carro, como "abs", "4 portas", e "step".

Acessando Propriedades de um Objeto

```
console.log(carro.marca);    // "Toyota"
console.log(carro.modelo);   // "Corolla"
console.log(carro.ano);      // 2024
console.log(carro.cor);      // "Prata"
console.log(carro.airbag);   // true
console.log(carro.itens);    // ["abs", "4 portas", "step"]
```

Para **acessar os valores de um objeto**, usamos a **notação de ponto** (.), seguida pelo nome da chave:

- carro.marca acessa o valor da propriedade marca, que é "Toyota",
- carro.modelo acessa o valor da propriedade modelo, que é "Corolla", e assim por diante.

Notação de Ponto: Essa é a forma mais comum de acessar as propriedades de um objeto.

Acessando Propriedades Usando Notação de Colchetes

```
console.log(carro["modelo"]); // "Corolla"  
console.log(carro);           // Exibe todo o objeto
```

- **Notação de Colchetes:** Também podemos acessar as propriedades de um objeto usando a **notação de colchetes** ([]). Nessa notação, a chave do objeto é passada como uma string:
 - carro["modelo"] acessa o valor da propriedade modelo do objeto carro.A **notação de colchetes** é útil quando a chave contém caracteres especiais ou espaços, ou quando queremos acessar uma propriedade dinamicamente (por exemplo, a partir de uma variável).

Modificando ou Adicionando Propriedades ao Objeto

```
carro.kmRodados = 15000;  
console.log(carro);
```

- **Adicionando Propriedade:** Podemos **adicionar novas propriedades** a um objeto a qualquer momento. No código acima, adicionamos a propriedade kmRodados ao objeto carro e atribuímos o valor 15000 a ela.
- O **console.log(carro)** exibe o objeto atualizado, que agora inclui a nova propriedade kmRodados.

Resumo dos Conceitos

- **Objeto:** No JavaScript, um objeto é uma coleção de **propriedades** (ou **chaves**) associadas a valores. Esses valores podem ser de qualquer tipo (strings, números, arrays, outros objetos, etc.).
- **Propriedade (ou chave):** É o nome que usamos para identificar um valor dentro de um objeto. Por exemplo, em carro.marca, a chave é marca.
- **Acesso a Propriedades:** Podemos acessar os valores de um objeto de duas maneiras:
 - **Notação de ponto** (carro.marca): A forma mais simples e direta.
 - **Notação de colchetes** (carro["marca"]): Útil para acessar propriedades dinamicamente ou quando a chave contém caracteres especiais.
- **Modificação e Adição de Propriedades:** Objetos podem ser modificados e podem ter novas propriedades adicionadas a qualquer momento, como demonstrado com carro.kmRodados = 15000.

Comparação entre Arrays e Objetos

- **Arrays:** Usam **índices numéricos** (começando do 0) para acessar os elementos. São ideais para armazenar **listas** de dados ordenados.
- **Objetos:** Usam **chaves** (que podem ser strings) para acessar os valores. São ideais para armazenar **informações estruturadas** com diferentes tipos de dados e propriedades.

Objetos em JavaScript são estruturas de dados poderosas e flexíveis, pois permitem armazenar diferentes tipos de dados em suas propriedades (chaves). Essa característica é um dos pontos fortes dos objetos, tornando-os ideais para representar **entidades** mais complexas no código.

Objetos Podem Armazenar Diversos Tipos de Dados

Ao contrário dos arrays, que geralmente armazenam uma sequência de dados do mesmo tipo, os objetos permitem que você armazene valores de **tipos diferentes** em suas propriedades. Isso significa que você pode associar uma string, um número, um booleano, um array ou até outro objeto como valor de uma chave no objeto.

Por exemplo, podemos ter um objeto livro que armazena não apenas o título (uma string), mas também o ano de publicação (número), o gênero (string), e até mesmo um número que indica a quantidade de páginas, como mostrado no exemplo abaixo:

```
let livro = {
    titulo: "Javascript para iniciantes", // string
    autor: "João Silva", // string
    ano: 2021, // número
    genero: "Programação", // string
};
```

Esse objeto livro contém diferentes tipos de dados associados a chaves que descrevem o livro. Cada chave, como titulo, autor, ano, e genero, representa uma característica dessa **entidade** (no caso, um livro).

Manipulando Propriedades de um Objeto

Adicionando Propriedades

Você pode adicionar novas propriedades ao objeto a qualquer momento, seja utilizando a **notação de ponto** ou a **notação de colchetes**. No exemplo abaixo, adicionamos a propriedade paginas e idioma ao objeto livro:

```
livro.paginas = 300;           // Adiciona a propriedade "paginas"  
livro["idioma"] = "Português"; // Adiciona a propriedade "idioma"
```

Removendo Propriedades

Você também pode remover propriedades de um objeto utilizando o operador delete. Se quisermos remover a propriedade idioma, fazemos assim:

```
delete livro.idioma; // Remove a propriedade "idioma"
```

Após a remoção, a chave idioma não estará mais presente no objeto.

Verificando Propriedades de um Objeto

Uma das operações úteis que podemos fazer com objetos é verificar se uma propriedade existe dentro dele. Isso pode ser feito com o operador `in`. Esse operador retorna `true` se a chave especificada existir no objeto, e `false` caso contrário. No exemplo abaixo:

```
console.log("autor" in livro); // true  
console.log("idioma" in livro); // false
```

- **"autor" in livro**: retorna `true`, porque o objeto `livro` contém a chave `autor`.
- **"idioma" in livro**: retorna `false`, porque a propriedade `idioma` foi removida anteriormente.

Objetos Como Representações de Entidades

Uma das principais razões para usar objetos em JavaScript é que eles são ideais para representar **entidades** no mundo real, como livros, pessoas, carros, etc. Cada chave no objeto é como uma **propriedade** dessa entidade, e o valor associado a essa chave é o valor ou característica correspondente.

Por exemplo, o objeto `livro` representa um livro específico, e suas chaves (`titulo`, `autor`, `ano`, `genero`) representam propriedades desse livro. Esses tipos de objetos podem ser usados para modelar **entidades** de forma muito eficaz.

Conclusão

- **Objetos** permitem armazenar diferentes tipos de dados (strings, números, arrays, objetos) em suas propriedades, o que os torna muito poderosos para modelar entidades do mundo real.
- **Propriedades** (ou chaves) de objetos podem ser acessadas ou modificadas, novas propriedades podem ser adicionadas, e propriedades existentes podem ser removidas.
- A **notação de ponto** e a **notação de colchetes** são formas de acessar e modificar as propriedades de objetos.
- O operador **in** é útil para verificar se uma propriedade existe em um objeto.

```
let livro = {  
    titulo: "Javascript para iniciantes",  
    autor: "João Silva",  
    ano: 2021,  
    genero: "Programação",  
};  
  
// Acessando as propriedades do objeto:  
console.log(livro.titulo); // "Javascript para iniciantes"  
console.log(livro.autor); // "João Silva"  
console.log(livro.ano); // 2021  
  
// Adicionando novas propriedades  
livro.paginas = 300;  
livro["idioma"] = "Português";  
console.log(livro); // Exibe o objeto com as novas propriedades  
  
// Removendo a propriedade "idioma"  
delete livro.idioma;  
console.log(livro); // Exibe o objeto sem a propriedade "idioma"  
  
// Verificando se a propriedade "autor" existe no objeto  
console.log("autor" in livro); // true  
console.log("idioma" in livro); // false
```

Manipulação de Tipos de Dados de Referência e Memória

Em Javascript, os **tipos de dados de referência** (como objetos e arrays) funcionam de maneira diferente dos **tipos primitivos** (como números, strings, booleanos). Quando manipulamos dados de referência, estamos na verdade manipulando **referências** para os dados na memória, não os próprios dados diretamente. Isso significa que alterações em uma variável que armazena uma referência podem afetar outras variáveis que compartilham a mesma referência.

Comportamento de Referências: Como Funciona a Memória?

Quando uma variável é atribuída a outra, no caso de **tipos de referência** (arrays e objetos), ambas as variáveis não guardam cópias independentes dos dados. Elas compartilham a mesma **referência** para o local na memória onde os dados estão armazenados. Isso implica que se você modificar o valor de um dado em uma das variáveis, a outra também será afetada, porque ambas apontam para o mesmo lugar na memória.

Exemplo 1: Manipulação de Arrays e Referência

```
let listaA = [1, 2, 3];
let listaB = listaA; // listaB agora aponta para o mesmo local na memória que listaA
listaB[0] = 99; // Modificando o primeiro valor de listaB

console.log(listaA); // [99, 2, 3]
console.log(listaB); // [99, 2, 3]
```

Explicação: Quando fazemos `let listaB = listaA;`, `listaB` e `listaA` não armazenam valores independentes, mas sim **referenciam o mesmo array**. Então, ao modificar o valor de `listaB[0]`, a alteração é refletida também em `listaA`, porque ambas as variáveis apontam para o mesmo array na memória.

Exemplo 2: Manipulação de Strings (Tipos Primitivos)

```
let string = "Olá";
let mensagem = string;
mensagem = "Olá Bem vindo";

console.log(mensagem); // "Olá Bem vindo"
console.log(string); // "Olá"
```

Explicação: Ao contrário dos arrays e objetos, **strings são tipos primitivos**, que são passados por valor e não por referência. Quando atribuímos `let mensagem = string;`, estamos criando uma cópia do valor de `string`. Isso significa que ao modificar `mensagem`, o valor de `string` não é afetado. Isso ocorre porque **tipos primitivos** (como strings e números) têm comportamento de **cópia por valor**.

Exemplo 3: Manipulação de Objetos

```
let objA = { nome: "Millene" };
let objB = objA; // objB agora aponta para o mesmo local na memória que objA
objB.idade = 34; // Modificando o valor de objB

console.log(objA); // { nome: "Millene", idade: 34 }
console.log(objB); // { nome: "Millene", idade: 34 }
```

Explicação: Similar ao exemplo de arrays, ao fazer let objB = objA;, objA e objB referenciam o mesmo objeto na memória. Então, quando adicionamos a propriedade idade em objB, a alteração também aparece em objA, pois ambos os nomes de variáveis compartilham a mesma referência para o mesmo objeto.

Exemplo 4: Criando uma Cópia de Referência

Para evitar a modificação do dado original quando manipulamos tipos de referência, podemos criar **cópias independentes** de arrays ou objetos. Isso pode ser feito utilizando técnicas de **desestruturação** ou **spread operator**.

```
let listaC = [...listaA]; // Criando uma cópia de listaA
console.log(listaC); // [1, 2, 3]
listaC[3] = 100; // Modificando a cópia, não afeta listaA
console.log(listaC); // [1, 2, 3, 100]
console.log(listaA); // [1, 2, 3]
```

Explicação: Aqui usamos o **spread operator (...)** para criar uma nova cópia de listaA. Agora, listaC é uma **cópia independente** de listaA, e a modificação em listaC não afeta listaA.

Exemplo 5: Criando uma Cópia de um Objeto

```
let objC = { ...objA }; // Criando uma cópia de objA
objC.corDeCabelo = "Castanho"; // Modificando a cópia

console.log(objC); // { nome: "Millene", corDeCabelo: "Castanho" }
console.log(objA); // { nome: "Millene" }
```

Explicação: Similar ao exemplo com arrays, ao usar o **spread operator (...)** em objetos, estamos criando uma nova **cópia independente** do objeto objA. A alteração em objC não afeta objA, porque eles agora são objetos distintos, com referências diferentes.

Resumo dos Conceitos

- **Tipos de Dados de Referência** (Arrays, Objetos):
 - Esses tipos de dados são manipulados por **referência**. Ou seja, quando você atribui um objeto ou array a uma nova variável, ambas as variáveis **referenciam o mesmo dado** na memória.
 - Modificações feitas em uma variável (que contém referência a um objeto ou array) afetam as outras variáveis que referenciam o mesmo dado.
- **Tipos Primitivos** (Strings, Números, Booleanos):
 - Esses tipos de dados são manipulados por **valor**. Ao atribuir uma variável a outra, uma **cópia do valor** é criada, de forma que alterações em uma variável não afetam a outra.
- **Cópias Independentes**:
 - Para evitar modificações indesejadas, você pode criar cópias independentes de arrays ou objetos utilizando o **spread operator (...)**. Isso cria uma nova referência para os dados, permitindo que você trabalhe com dados separados.

Conclusão

Ao trabalhar com tipos de referência, é importante entender como **referências e memória** funcionam em JavaScript. Se você precisar de cópias independentes para evitar alterações indesejadas, usar o **spread operator** ou métodos como `Object.assign()` pode ser uma solução eficaz. Já os tipos primitivos, por serem manipulados por valor, oferecem maior controle e previsibilidade, pois modificações em uma variável não afetam as demais.

Módulo 4

FUNÇÕES

FUNÇÕES

FUNÇÕES



Uma **função** em JavaScript é um bloco de código reutilizável que executa uma tarefa específica. Ela pode receber entradas (chamadas de parâmetros) e retornar uma saída (um valor), ou simplesmente realizar uma ação. O principal benefício de usar funções é a **reutilização de código**, permitindo que você execute a mesma ação múltiplas vezes sem precisar reescrever o código.

Em termos simples, uma função é uma **caixa de operações** que recebe **entradas**, realiza algum **processamento** e, em alguns casos, retorna uma **saída**. Se não retornar um valor, a função pode apenas realizar ações, como exibir algo no console ou alterar valores.

Funções são Tipos de Dados de Referência

Em JavaScript, funções são **objetos** e, como tal, são **tipos de dados de referência**. Isso significa que elas podem ser atribuídas a variáveis, passadas como argumentos para outras funções e até retornadas de outras funções. Quando você manipula uma função, você está manipulando uma **referência** a ela, não uma cópia.

Exemplo de Função: Saudações para Usuários de um Site

Vamos imaginar que você tem um site de login e precisa saudar diferentes usuários, como "Daniel", "Lira", "Millene", "Alon" e "Fred". Para evitar escrever o mesmo código repetidamente, você pode criar uma função chamada saudacao, que receberá o nome do usuário e exibirá uma mensagem de boas-vindas no console.

```
// Função saudacao que recebe o parâmetro "nome"
function saudacao(nome) {
    console.log("Olá " + nome + ", bem-vindo ao site!");
}
```

```
// Chamando a função com diferentes nomes
saudacao("Daniel"); // Saída: Olá Daniel, bem-vindo ao site!
saudacao("Lira"); // Saída: Olá Lira, bem-vindo ao site!
saudacao("Millene"); // Saída: Olá Millene, bem-vindo ao site!
saudacao("Alon"); // Saída: Olá Alon, bem-vindo ao site!
saudacao("Fred"); // Saída: Olá Fred, bem-vindo ao site!
```

Explicação do Código:

- **Definição da Função:**
 - A função saudacao(nome) é definida para receber um parâmetro chamado nome.
 - Dentro da função, usamos console.log para imprimir a mensagem de boas-vindas, concatenando o valor do parâmetro nome com o texto "Olá, bem-vindo ao site!".
- **Chamada da Função:**
 - Cada vez que chamamos a função saudacao(), passamos um valor diferente para o parâmetro nome (como "Daniel", "Lira", "Millene", etc.).
 - A função então utiliza o valor passado e exibe a mensagem personalizada no console.

Por que as Funções São Úteis?

No exemplo acima, sem uma função, você precisaria escrever o código console.log("Olá nome, bem-vindo ao site!") várias vezes para cada usuário. Isso não só é repetitivo, mas também aumenta o risco de erro, caso você precise alterar a mensagem em algum momento.

Ao criar uma função, você centraliza a lógica em um único lugar, tornando o código mais **organizado, manutenível e fácil de entender**. Além disso, como uma função é **reutilizável**, você pode chamar saudacao() quantas vezes precisar, com diferentes parâmetros, sem reescrever o mesmo código.

Funções e Tipos de Referência

Como mencionado anteriormente, funções em JavaScript são **objetos**, ou seja, são **tipos de referência**. Isso significa que, em vez de apenas passar o código da função, você está, de fato, passando a referência para essa função. Você pode passar a função como argumento para outra função, armazená-la em variáveis ou até retorná-la de outras funções.

Conclusão

Funções são blocos de código poderosos em JavaScript que ajudam a organizar e reutilizar o código de maneira eficiente. Elas são especialmente úteis quando você precisa realizar a mesma tarefa várias vezes com diferentes entradas, como no exemplo de saudar diferentes usuários em um site. Além disso, por serem tipos de referência, elas podem ser manipuladas de diversas maneiras, oferecendo flexibilidade no desenvolvimento do seu código.

Como Criar e Utilizar Funções em JavaScript

Em JavaScript, as funções são uma maneira poderosa de organizar e reutilizar código. Elas podem ser criadas para executar tarefas específicas e podem receber parâmetros para tornar o código mais dinâmico. A seguir, vamos explorar como criar funções e usá-las com base em um exemplo prático.

Sintaxe Básica de Função

A sintaxe básica de uma função em JavaScript segue o seguinte formato:

```
function nomeDaFuncao() {  
    // instruções  
}
```

- **function** é a palavra-chave usada para declarar a função.
- **nomeDaFuncao** é o nome que você escolhe para a função.
- Dentro das chaves {}, você coloca as instruções que a função executará.

Exemplo 1: Função Simples - Enviar Mensagem

Vamos criar uma função que imprime uma mensagem no console.

```
// Função simples para enviar uma mensagem
function enviarMensagem() {
    console.log("Para continuar você precisa informar o seu nome para cadastro");
}

// Chamando a função
enviarMensagem(); // Executa a função e imprime a mensagem
```

- **O que acontece aqui?**

- A função enviarMensagem é definida para mostrar uma mensagem no console.
- Depois, chamamos a função usando enviarMensagem() para que a mensagem seja exibida.

Exemplo 2: Função com Parâmetros - Cadastro de Usuário

Agora, vamos criar uma função que recebe informações do usuário, como nome e sobrenome, e exibe uma mensagem personalizada.

```
// Função para cadastrar um usuário
function cadastrar(nome, sobrenome) {
    console.log(`Olá ${nome} ${sobrenome}, você foi cadastrado com sucesso`);
}

// Chamando a função com parâmetros
cadastrar("Daniel", "Porto");
```

O que acontece aqui?

- A função cadastrar recebe dois parâmetros: nome e sobrenome.
- Usamos esses parâmetros para construir uma mensagem de boas-vindas personalizada.
- Quando chamamos a função com os valores "Daniel" e "Porto", a mensagem "Olá Daniel Porto, você foi cadastrado com sucesso" é exibida no console.

Exemplo 3: Função com Retorno - Operações Matemáticas

Agora, vamos criar uma função que executa uma operação matemática e retorna um valor.

```
// Função para calcular saldo de banco
function banco(deposito, saque) {
    let saldo = deposito - saque;
    return saldo;
}

// Chamando a função e armazenando o retorno
let saldoAtual = banco(10000, 780);
console.log(`O saldo atual da sua conta é de ${saldoAtual} reais`);
```

O que acontece aqui?

- A função banco recebe dois parâmetros: deposito e saque.
- Ela calcula o saldo subtraindo o valor do saque do depósito e retorna esse valor.
- Quando chamamos a função, o valor retornado é armazenado na variável saldoAtual, que depois é exibida no console.

Exemplo 4: Função Principal - Organizando Funções

Agora, vamos criar uma função principal (main) que chama todas as outras funções para realizar um processo completo.

```
// Função principal que organiza a execução de outras funções
function main() {
    enviarMensagem(); // Chama a função de enviar mensagem
    cadastrar("Daniel", "Porto"); // Chama a função de cadastro
    let saldo = banco(10000, 780); // Chama a função de banco e armazena o saldo
    console.log(`O saldo atual da sua conta é de ${saldo} reais`); // Exibe o saldo
}

// Chamando a função principal
main();
```

- **O que acontece aqui?**

- A função main organiza a execução das outras funções:
 - Chama enviarMensagem para mostrar a mensagem inicial.
 - Chama cadastrar com o nome e sobrenome do usuário.
 - Chama banco para calcular o saldo e armazená-lo em saldo.
 - Exibe o saldo atual no console.

Conceitos:

- **Função Simples:** Uma função sem parâmetros que executa uma ação (como enviarMensagem).
- **Função com Parâmetros:** Uma função que recebe dados de entrada (como cadastrar), permitindo que o comportamento da função seja dinâmico.
- **Função com Retorno:** Funções que retornam um valor após executar uma operação (como banco).
- **Função Principal:** Uma função que organiza o fluxo de execução do programa, chamando outras funções conforme necessário (como main).

Conclusão:

As funções são uma maneira poderosa de estruturar seu código e torná-lo mais organizado e reutilizável. Com as funções, você pode dividir tarefas complexas em partes menores e mais fáceis de entender. Além disso, as funções ajudam a evitar a repetição de código, o que torna seu programa mais eficiente e fácil de manter.

Como Chamar uma Função em JavaScript

Em JavaScript, **chamar uma função** é essencial para que o código dentro dela seja executado. Quando uma função é definida, ela não é executada automaticamente; você precisa "chamá-la" para que as instruções dentro dela sejam executadas.

Sem a chamada, a função simplesmente existe, mas não faz nada.

Sintaxe para Chamar uma Função

Para chamar uma função, você usa o nome dela seguido de parênteses (). Dentro dos parênteses, você pode passar valores (argumentos) que a função utilizará, se ela estiver configurada para receber parâmetros.

Exemplo 1: Chamando uma Função para Exibir Detalhes

Vamos entender isso com um exemplo onde temos uma função que exibe detalhes sobre professores e cursos.

```
// Definindo a função que exibe detalhes do professor e do curso
function exibirDetalhes(nome, curso) {
    console.log("Professor: " + nome + " Curso: " + curso); // Executa a instrução
}

// Chamando a função com diferentes argumentos
exibirDetalhes("Lira", "Python");      // Passa "Lira" e "Python"
exibirDetalhes("Daniel", "Javascript"); // Passa "Daniel" e "Javascript"
exibirDetalhes("Alon", "Power BI");     // Passa "Alon" e "Power BI"
exibirDetalhes("Fred", "Excel");        // Passa "Fred" e "Excel"
```

- **O que acontece no código?**

- A função exibirDetalhes foi definida para receber dois parâmetros: nome e curso.
- Dentro da função, usamos console.log para imprimir essas informações.
- Para que a função execute, a chamamos **com os valores desejados**: "Lira" e "Python", "Daniel" e "Javascript", etc.
- **Se não chamássemos a função**, nada seria impresso no console.

Exemplo 2: Chamando uma Função Simples

Vamos ver agora uma função simples, que não recebe parâmetros e apenas imprime uma mensagem.

```
// Definindo a função mensagem
function mensagem() {
    console.log("Imprimindo uma mensagem!"); // Exibe uma mensagem
}

// Chamando a função múltiplas vezes
mensagem(); // Primeira chamada
mensagem(); // Segunda chamada
mensagem(); // Terceira chamada
mensagem(); // Quarta chamada
mensagem(); // Quinta chamada
```

O que acontece no código?

- A função mensagem foi definida para simplesmente exibir uma mensagem no console.
- **Cada vez que chamamos mensagem()**, o código dentro dela é executado e a mensagem é exibida.
- **Sem chamar a função**, a mensagem não será impressa, mesmo que a função exista no código.

Por que Chamar a Função é Importante?

- **Execução Controlada:** As funções em JavaScript não são executadas automaticamente quando você as define. Você precisa chamar explicitamente a função para que o código dentro dela seja executado.
- **Reutilização:** Uma vez que a função está definida, você pode chamá-la quantas vezes precisar, com diferentes argumentos, sem precisar reescrever o código.
- **Organização:** As funções permitem que você organize seu código em "blocos" reutilizáveis, tornando o código mais limpo e fácil de entender.

Conclusão

- Em JavaScript, **chamar uma função** é essencial para que ela execute suas instruções. Sem a chamada, a função existe, mas não realiza nenhuma ação. O uso de chamadas de função torna seu código mais modular e reutilizável, facilitando a organização e a execução de tarefas complexas em seu programa.

Em JavaScript, parâmetros e argumentos são conceitos essenciais para entender como as funções recebem e utilizam valores. Vamos desmembrar esses conceitos e ver como eles funcionam.

1. O que são Parâmetros?

- Parâmetros são os nomes utilizados dentro de uma função para representar os valores que ela irá receber. Eles são definidos quando você cria a função e servem como "variáveis locais" que armazenam os valores passados para a função.
- Definição de parâmetros: Os parâmetros são listados entre os parênteses da declaração da função, logo após o nome da função.

```
function soma(numero1, numero2) { // 'numero1' e 'numero2' são parâmetros
    console.log(numero1 + numero2);
}
```

Quando você define a função soma, você está criando parâmetros chamados numero1 e numero2. Eles são usados dentro da função para receber valores e executar a lógica (neste caso, somar os números).

2. O que são Argumentos?

Argumentos são os **valores reais** que você passa para a função quando a chama. Esses valores correspondem aos parâmetros definidos anteriormente.

- **Passando argumentos:** Quando você chama uma função, você fornece **argumentos** que são atribuídos aos parâmetros na função.

```
soma(10, 5); // '10' e '5' são argumentos passados para os parâmetros numero1 e numero2
```

- Quando a função soma(10, 5) é chamada, os valores 10 e 5 são passados como **argumentos** para os **parâmetros** numero1 e numero2.
- A função então executa a soma e imprime 15 no console.

Exemplo Prático com Mais de um Parâmetro e Argumentos

Aqui está outro exemplo, onde calculamos o preço total de um item:

```
function calcularPrecoTotal(precoUnitario, quantidade) { // 'precoUnitario' e 'quantidade' são parâmetros
  let total = precoUnitario * quantidade;
  console.log("O total da sua compra é: " + total);
}

let camiseta = 30;           // Variáveis definidas
let quantidadeItem = 3;     // Variáveis definidas

calcularPrecoTotal(camiseta, quantidadeItem); // 'camiseta' e 'quantidadeItem' são argumentos passados
```

- A função calcularPrecoTotal foi definida com dois parâmetros: precoUnitario e quantidade.
- Quando a função é chamada com os argumentos camiseta e quantidadeItem, esses valores são atribuídos aos parâmetros, permitindo que a função calcule o total da compra.

Diferença entre Parâmetros e Argumentos

- **Parâmetros** são variáveis nomeadas que aparecem na definição da função.
- **Argumentos** são os valores reais que você passa para a função quando a chama.

Resumo do Fluxo:

- **Definição da Função:**
 - Na definição de uma função, você cria **parâmetros** que são como "caixas" esperando receber valores.
- **Chamada da Função:**
 - Quando você chama a função, você fornece **argumentos** (valores reais) que são atribuídos aos parâmetros da função.
- **Execução da Função:**
 - A função então executa sua lógica utilizando os valores que foram passados como argumentos.

Conclusão

Entender a diferença entre parâmetros e argumentos é essencial para trabalhar com funções em JavaScript. **Parâmetros** são definidos na criação da função, e **argumentos** são os valores que você passa para a função quando a chama. Esse conceito é importante porque permite que você reutilize funções com diferentes entradas, tornando seu código mais modular e flexível.

Ao criar funções no JavaScript, elas podem realizar diversas tarefas, como processar informações, executar cálculos e até interagir com APIs. Mas, às vezes, precisamos que a função **devolva** um valor para que possamos usá-lo fora dela. É aí que entra o conceito de **retorno** de valores.

Uma função pode ter um valor retornado usando a palavra-chave `return`. Isso permite:

- **Encerrar a execução da função:** Assim que o `return` é executado, nenhum código abaixo dele será lido.
- **Passar um valor para o código que chamou a função:** Esse valor pode ser armazenado em uma variável, usado diretamente ou até passado como argumento para outra função.

Exemplo Explicativo: Calculando o Valor Total de um Pedido

Vamos utilizar o exemplo abaixo para entender como retornar valores de uma função:

```
let pedido = {  
    id: 1234,  
    nome: "João",  
    email: "joao@example.com",  
    lanche: 12,  
    batataFrita: 6,  
    suco: 4,  
};
```

```
// Função para processar o pedido  
  
function processarPedido(id, item1, item2, item3) {  
    let totalPedido = item1 + item2 + item3; // Calcula o total do pedido  
    console.log("Pedido: " + id + " Processando");  
    return totalPedido; // Retorna o total do pedido  
    // Código abaixo de "return" não será executado  
}
```

```
// Chamando a função e armazenando o retorno  
let retornoDaFuncao = processarPedido(  
    pedido.id,  
    pedido.lanche,  
    pedido.batataFrita,  
    pedido.suco  
);  
  
console.log("O total do pedido é: " + retornoDaFuncao);
```



Criamos um objeto pedido com os dados de um pedido, incluindo itens e preços.

- A função processarPedido foi definida para calcular o valor total dos itens e retornar o resultado.
- A soma dos valores item1, item2 e item3 é armazenada na variável totalPedido.
- Usamos o comando return totalPedido para devolver o valor calculado.
- Ao chamar a função, armazenamos o valor retornado na variável retornoDaFuncao para usá-lo posteriormente.

Resultado no Console

```
Pedido: 1234 Processando
O total do pedido é: 22
```

- A função processou o pedido, calculou o valor total e retornou o resultado.
- O valor retornado foi exibido no console com a mensagem "O total do pedido é: 22".

Comparação com Funções Sem Retorno

Para fins de comparação, veja a função enviarNotificacao:

```
function enviarNotificacao(nome, idPedido, email) {
    console.log(`Enviando email para ${email} confirmando o pedido do número ${idPedido}`);
};

console.log(`Mensagem: ${nome}, pedido confirmado`);

}
```

Esta função realiza uma tarefa (simula o envio de uma notificação), mas **não retorna nenhum valor**. O resultado está apenas nos `console.log`. Se você tentasse armazenar a chamada dessa função em uma variável, como:

```
let resultado = enviarNotificacao(pedido.nome, pedido.id, pedido.email);
console.log(resultado);
```

O valor exibido seria `undefined`, pois a função não tem um `return`.

- **Funções com retorno:** São ideais quando você precisa de um resultado que será usado posteriormente no código.
- **Funções sem retorno:** São úteis para realizar ações específicas, como imprimir no `console` ou modificar algo diretamente.

Sempre que criar uma função, pergunte-se: "**Preciso de um resultado que será reutilizado ou só quero realizar uma ação?**" Isso ajudará a decidir se o `return` é necessário.



Identificação de tipo

Ao passar o cursor em cima de uma variável, o VS Code consegue identificar a qual tipo de dados essa variável tem seu valor atribuído. No exemplo ao lado, o programa mostra que o valor armazenada é do tipo number.

```
js aula3_3.js > ...
1 function resolverBhaskara() {}
2
3 function calcularRaizQuadrada(base) {
4   return base ** (1 / 2);
5 }
6 let valorRaizQuadrada: number
7 let valorRaizQuadrada = calcularRaizQuadrada(4);
```

Antes de explorarmos mais profundamente os múltiplos parâmetros em funções, vamos revisar alguns conceitos importantes para garantir que você compreenda bem o que está acontecendo:

- **Parâmetros:** São os nomes que definimos entre parênteses na **declaração** de uma função. Eles servem como "espaços reservados" para receber valores quando a função for chamada.
- **Argumentos:** São os valores que fornecemos quando **chamamos** ou **executamos** a função. Esses valores são passados para preencher os parâmetros definidos.
- **return:** É utilizado no final de uma função para devolver um valor como resultado. Isso permite que o valor retornado seja usado em outras partes do código e, ao mesmo tempo, encerra a execução da função.

Trabalhando com Múltiplos Parâmetros

Uma função pode aceitar múltiplos parâmetros, separados por vírgulas. Quando isso acontece, a **ordem dos argumentos** ao chamar a função deve corresponder exatamente à **ordem dos parâmetros** definidos.

Considere o exemplo ao lado:

```
let pedido = {  
    id: 1234,  
    nome: "João",  
    email: "joao@example.com",  
    lanche: 12,  
    batataFrita: 6,  
    suco: 4,  
};  
  
// Função para processar o pedido  
function processarPedido(id, item1, item2, item3) {  
    let totalPedido = item1 + item2 + item3; // Calcula o total do pedido  
    console.log("Pedido: " + id + " Processando");  
    return totalPedido; // Retorna o total do pedido  
}  
  
// Chamando a função e passando os argumentos na ordem correta  
let retornoDaFuncao = processarPedido(  
    pedido.id,           // Argumento para o parâmetro "id"  
    pedido.lanche,       // Argumento para o parâmetro "item1"  
    pedido.batataFrita, // Argumento para o parâmetro "item2"  
    pedido.suco         // Argumento para o parâmetro "item3"  
);  
  
console.log("O total do pedido é: " + retornoDaFuncao);
```

Entendendo o Código

A função `processarPedido` foi definida com **quatro parâmetros**:

- `id`: para identificar o pedido.
- `item1`, `item2`, e `item3`: para receber os valores dos itens do pedido.

Na chamada da função, passamos **quatro argumentos** que correspondem, na mesma ordem, aos parâmetros definidos:

- `pedido.id` é passado para o parâmetro `id`.
- `pedido.lanche` é passado para o parâmetro `item1`.
- `pedido.batataFrita` é passado para o parâmetro `item2`.
- `pedido.suco` é passado para o parâmetro `item3`.

Importância da Ordem dos Argumentos

A ordem em que passamos os argumentos é **crucial**. O JavaScript associa os valores aos parâmetros com base na ordem em que aparecem.

Se trocarmos a ordem dos argumentos ao chamar a função, os resultados podem ser incorretos. Por exemplo:

```
let retornoErrado = processarPedido(  
    pedido.id,  
    pedido.suco,      // Aqui o suco foi passado como "item1"  
    pedido.lanche,   // O Lanche foi passado como "item2"  
    pedido.batataFrita // E a batata frita como "item3"  
);  
  
console.log("Resultado errado: " + retornoErrado);
```

Neste caso, o cálculo total será feito de forma incorreta, porque os valores não correspondem aos itens esperados.

Um Segundo Exemplo: Notificações

Outra aplicação útil de múltiplos parâmetros é a função enviarNotificacao, que simula o envio de um email. Veja:

```
function enviarNotificacao(nome, idPedido, email) {  
    console.log(`Enviando email para ${email} confirmando o pedido do número ${idPedido}`);  
    console.log(`Mensagem: ${nome}, pedido confirmado`);  
}
```

Aqui, temos três parâmetros: nome, idPedido, e email. Quando chamamos a função, devemos passar os argumentos na ordem correta:

```
enviarNotificacao(pedido.nome, pedido.id, pedido.email);
```

O resultado será:

```
Enviando email para joao@example.com confirmando o pedido do número 1234  
Mensagem: João, pedido confirmado
```

Se a ordem for alterada, o email pode ser enviado de forma errada, ou o conteúdo da mensagem pode não fazer sentido.

Boas Práticas ao Trabalhar com Múltiplos Parâmetros

- **Defina parâmetros claros e significativos:** Use nomes que representem bem o propósito dos dados que eles devem receber.
- **Mantenha a ordem consistente:** Certifique-se de passar os argumentos na mesma ordem dos parâmetros definidos.
- **Use objetos como argumentos, se necessário:** Para funções que precisam de muitos parâmetros, passar um objeto como argumento pode ser mais organizado. Por exemplo:

```
function processarPedido({ id, lanche, batataFrita, suco }) {  
  let totalPedido = lanche + batataFrita + suco;  
  console.log("Pedido: " + id + " Processando");  
  return totalPedido;  
}  
  
let total = processarPedido(pedido);  
console.log("O total do pedido é: " + total);
```

Conclusão

- **Parâmetros** são placeholders definidos na função.
- **Argumentos** são os valores que preenchem os parâmetros na chamada da função.
- A **ordem dos argumentos** deve corresponder à ordem dos parâmetros para evitar problemas.
- O uso de `return` permite que a função devolva um resultado e encerre sua execução.

Com esses conceitos claros, você estará pronto para criar funções mais complexas e organizadas, sempre garantindo que os valores corretos sejam usados no momento certo.



O JavaScript nos permite criar e usar funções de maneiras diferentes. Uma dessas maneiras é através de **declarações de função** e **expressões de função**. Ambas são úteis em diferentes contextos, mas têm diferenças importantes que precisam ser entendidas. Nesta aula, exploraremos os conceitos e as diferenças entre esses dois formatos.

Declaração de Função

Uma **declaração de função** cria uma função com um nome definido. Ela tem algumas características específicas:

- **Pode ser chamada antes de sua definição no código:** Isso é possível graças ao mecanismo chamado **hoisting**, que eleva a definição das funções declaradas para o topo do escopo.
- **Tem um nome associado:** Esse nome é obrigatório e usado para referenciar a função.

Exemplo de Declaração de Função

```
function soma(a, b) {  
    return a + b;  
}  
  
let total = soma(3, 4) + 10; // Chamando a função antes de usá-la  
console.log(total); // Saída: 17  
  
let valor = soma(2, 4); // Chamando a função após sua definição  
console.log(valor); // Saída: 6
```

- A função soma é declarada usando a palavra-chave function, seguida do nome soma e de uma lista de parâmetros (a e b).
- Quando chamamos soma(3, 4), os valores 3 e 4 são passados como **argumentos** para os parâmetros a e b. A função retorna a soma desses valores, que é 7.
- Como é uma **declaração de função**, podemos chamá-la antes mesmo de ela aparecer no código. Isso funciona devido ao **hoisting**.

Expressão de Função

Uma **expressão de função** é uma função que é atribuída a uma variável ou constante. Isso significa que ela não tem um nome próprio (embora possa ter, em casos específicos) e só pode ser usada **após** ser definida.

- **Não pode ser chamada antes de sua definição:** O hoisting não se aplica da mesma forma às expressões de função.
- **Pode ou não ter um nome:** Se tiver um nome, ele é usado apenas para referência dentro da própria função.

Exemplo de Expressão de Função

```
let total = function soma(a, b) {
    return a + b;
};

console.log(total(3, 4)); // Saída: 7
console.log(total(13, 4)); // Saída: 17
console.log(total(31, 14)); // Saída: 45
console.log(total(3, 24)); // Saída: 27
```

- Criamos uma **expressão de função** e a atribuímos à variável total.
- Embora a função tenha um nome (soma), ele é opcional. Aqui, o nome é útil apenas dentro da função, mas podemos omiti-lo e usar uma função **anônima**.
- Como é uma expressão de função, ela só pode ser chamada **depois** de sua definição no código.

Diferenças Entre Declaração e Expressão de Função

Característica	Declaração de Função	Expressão de Função
Definição	Usa <code>function nome(...){ ... }</code>	Usa <code>let/const nome = function(...){ ... }</code>
Hoisting	É elevada ao topo do escopo	Não é elevada
Necessidade de um nome	Nome obrigatório	Nome opcional (pode ser anônima)
Quando pode ser chamada	Antes ou depois da definição	Apenas depois da definição

Usando Expressões de Função

Expressões de função são particularmente úteis quando queremos passar funções como argumentos ou armazená-las em variáveis.

Por exemplo:

Exemplo com Função Anônima

Aqui, a função não tem nome (é anônima) e foi atribuída à variável `total`. A função pode ser chamada normalmente usando `total`.

```
let total = function (a, b) {  
    return a + b;  
};  
  
console.log(total(3, 4)); // Saída: 7
```

Quando Usar Cada Tipo?

- **Declarações de função:** Use quando você precisa de uma função reutilizável e quer a flexibilidade de chamá-la em qualquer parte do código (graças ao hoisting).
- **Expressões de função:** Use quando deseja atribuir funções a variáveis, criar funções anônimas, ou quando precisa de funções como argumentos para outras funções.

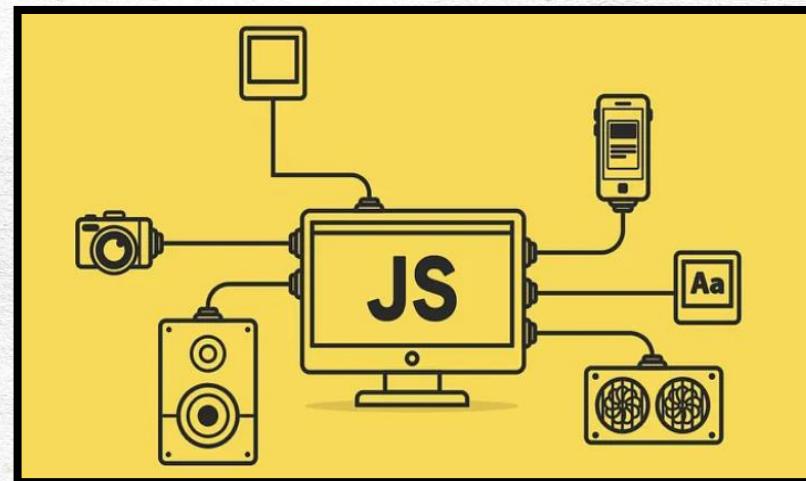
As funções são uma das principais características do JavaScript, permitindo que criemos códigos dinâmicos e reutilizáveis. Entre os conceitos mais poderosos estão as **funções de alta ordem**. Nesta aula, vamos entender o que elas são, como funcionam, e como aplicá-las no dia a dia da programação.

O Que São Funções de Alta Ordem?

Uma **função de alta ordem** é uma função que:

- **Recebe outra função como argumento.**
- **Retorna uma nova função como resultado.**

Esse comportamento permite que você trate funções como dados, tornando o código mais modular, reutilizável e elegante.



Exemplo 1: Funções como Argumentos

No exemplo abaixo, a função aplicarOperacao é uma função de alta ordem porque recebe outra função (operacao) como um dos seus argumentos:

- **Definição da Função de Alta Ordem:**
 - A função aplicarOperacao aceita dois parâmetros:
 - x: o número sobre o qual a operação será realizada.
 - operacao: a função que será aplicada ao número x.
- **Função Auxiliar:**
 - A função dobrar recebe um número como argumento e retorna o dobro desse número.
- **Uso da Função de Alta Ordem:**
 - Chamamos aplicarOperacao(5, dobrar). Aqui:
 - 5 é o valor de x.
 - dobrar é a função que será aplicada a 5.
 - O resultado é o retorno da função dobrar(5), que é 10.

Esse exemplo mostra como funções podem ser passadas como argumentos para tornar o código mais dinâmico.

```
function aplicarOperacao(x, operacao) {  
    // Função de Alta Ordem  
    return operacao(x);  
}  
  
function dobrar(numero) {  
    return numero * 2;  
}  
  
const resultado = aplicarOperacao(5, dobrar);  
console.log(resultado); // Saída: 10
```

Exemplo 2: Funções como Resultado

No próximo exemplo, criamos uma função de alta ordem chamada criarIncrementador. Ela retorna uma nova função que pode ser usada posteriormente:

```
function criarIncrementador(incremento) {  
    // Função de Alta Ordem  
    return function (numero) {  
        return numero + incremento;  
    };  
  
const incrementoPor2 = criarIncrementador(2);  
console.log(incrementoPor2(5)); // Saída: 7
```

Definição da Função de Alta Ordem:

- A função criarIncrementador aceita um valor chamado incremento.
- Ela retorna uma nova função anônima que soma o valor de incremento a um número passado como argumento.

Criação da Função Incrementadora:

- Chamamos criarIncrementador(2), que retorna uma nova função:

```
function (numero) {  
    return numero + 2;  
}
```

Uso da Função Retornada:

- Chamamos incrementoPor2(5). Aqui:
 - 5 é o argumento para a função retornada.
 - A função retorna $5 + 2$, ou seja, 7.
- Essa nova função é atribuída à constante incrementoPor2.

Por Que Usar Funções de Alta Ordem?

As funções de alta ordem são úteis porque:

- **Tornam o código mais modular:**
 - Você pode criar funções pequenas e reutilizáveis para realizar tarefas específicas.
- **Permitem abstração:**
 - Você pode criar funções genéricas que aceitam diferentes comportamentos como argumento ou resultado.
- **Facilitam o uso de padrões funcionais:**
 - Isso é muito comum em bibliotecas como React, ou métodos como map, filter e reduce.

Outros Exemplos Práticos

Usando Funções Anônimas como Argumentos

Você pode passar diretamente uma função anônima como argumento para uma função de alta ordem:

```
const resultado = aplicarOperacao(5, function (numero) {
    return numero ** 2; // Eleva ao quadrado
});

console.log(resultado); // Saída: 25
```

Funções de Alta Ordem em Métodos Nativos

Métodos como map, filter e reduce são exemplos de funções de alta ordem em JavaScript. Por exemplo:

```
const numeros = [1, 2, 3, 4];

// Usando 'map' com uma função como argumento
const dobrados = numeros.map(function (numero)
    return numero * 2;
);

console.log(dobrados); // Saída: [2, 4, 6, 8]
```

Diferenças Entre Funções de Alta Ordem e Funções Comuns

Característica	Função Comum	Função de Alta Ordem
Receber Funções como Argumento	Não aceita	Aceita uma ou mais funções
Retornar Outra Função	Não retorna	Pode retornar uma nova função
Exemplo	<pre>function soma(a, b) { ... }</pre>	<pre>function aplicarOperacao(x, f) { ... }</pre>

As funções de alta ordem permitem que você escreva códigos mais dinâmicos e reutilizáveis, aproveitando a flexibilidade das funções como "primeira classe" no JavaScript. Entender como elas funcionam é um passo importante para dominar conceitos mais avançados de programação funcional e criar soluções elegantes e eficientes.

Em Javascript, funções podem ter **parâmetros opcionais**, o que significa que nem todos os argumentos precisam ser fornecidos ao chamar a função. Para lidar com casos em que um argumento não é fornecido, podemos usar **valores padrão**. Isso torna nosso código mais robusto e previsível.

O Que São Parâmetros Opcionais?

Parâmetros opcionais são aqueles que não precisam ser obrigatoriamente definidos ao chamar uma função. Sem eles, as funções podem gerar erros ou retornar valores inesperados quando argumentos não são fornecidos.

Por exemplo:

```
function cumprimentar(saudacao, nome) {  
    console.log(`Olá ${nome}, ${saudacao}`);  
}  
  
cumprimentar("Boa tarde"); // Saída: "Olá undefined, Boa tarde"
```

O Problema

No exemplo acima, o parâmetro nome não foi fornecido, então o JavaScript definiu seu valor como undefined. Isso pode causar comportamentos inesperados.

Solução: Valores Padrão

Os **valores padrão** permitem que você defina um valor inicial para um parâmetro, caso ele não seja fornecido na chamada da função. Isso é feito diretamente na definição da função, usando o operador =.

Exemplo com Valor Padrão

```
function cumprimentar(saudacao, nome = "visitante") {  
    console.log(`Olá ${nome}, ${saudacao}`);  
}  
  
cumprimentar("Boa tarde"); // Saída: "Olá visitante, Boa tarde"  
cumprimentar("Bom dia", "João"); // Saída: "Olá João, Bom dia"
```

- A função cumprimentar foi definida com dois parâmetros:
- saudacao (obrigatório).
- nome (opcional, com valor padrão "visitante").
- Quando chamamos a função sem passar o argumento nome, o valor padrão "visitante" é usado.
- Se fornecermos um valor para nome, ele substitui o valor padrão.

Exemplos Mais Avançados

Combinação de Valores Padrão e Lógica Interna

Você pode usar valores padrão junto com lógica adicional dentro da função:

```
function calcularPreco(precoBase, desconto = 0) {  
    let precoFinal = precoBase - desconto;  
    return precoFinal;  
}  
  
console.log(calcularPreco(100)); // Saída: 100 (desconto padrão de 0)  
console.log(calcularPreco(100, 20)); // Saída: 80
```

Valores Dinâmicos nos Parâmetros

Os valores padrão também podem ser dinâmicos, baseados em outras variáveis ou expressões:

Aqui, se o parâmetro role não for passado, ele assume o valor padrão "usuário padrão".

```
function criarUsuario(nome, role = "usuário padrão") {
    console.log(`Nome: ${nome}, Role: ${role}`);
}

criarUsuario("Ana"); // Saída: "Nome: Ana, Role: usuário padrão"
criarUsuario("Carlos", "administrador"); // Saída: "Nome: Carlos, Role: administrador"
```

Cuidados com Valores Padrão

Ordem dos Parâmetros

É importante lembrar que a ordem dos argumentos fornecidos deve corresponder à ordem dos parâmetros na definição da função. Por exemplo:

```
function cumprimentar(nome = "visitante", saudacao = "Olá") {
    console.log(`${saudacao}, ${nome}!`);
}

cumprimentar(); // Saída: "Olá, visitante!"
cumprimentar("João"); // Saída: "Olá, João!"
cumprimentar("Maria", "Bom dia"); // Saída: "Bom dia, Maria!"
```

Valor Padrão como undefined

Se você passar explicitamente undefined como argumento, o valor padrão ainda será aplicado:

```
function cumprimentar(nome = "visitante") {  
  console.log(`Olá, ${nome}!`);  
}  
  
cumprimentar(undefined); // Saída: "Olá, visitante!"
```

Diferença Entre Valor Padrão e Lógica Interna

Usar valores padrão diretamente nos parâmetros torna o código mais limpo e fácil de entender, em vez de verificar manualmente se um parâmetro foi passado:

Lógica Interna (Sem Valor Padrão)

```
function cumprimentar(nome) {  
  if (!nome) {  
    nome = "visitante";  
  }  
  console.log(`Olá, ${nome}!`);  
}
```



Com Valor Padrão

```
function cumprimentar(nome = "visitante") {  
    console.log(`Olá, ${nome}!`);  
}
```

A segunda abordagem é mais simples e direta.

Os **parâmetros opcionais** e os **valores padrão** tornam as funções mais flexíveis e confiáveis. Ao definir um valor padrão, você evita problemas causados por argumentos ausentes e cria um código mais previsível.

Dicas Finais

- Use **valores padrão** sempre que um parâmetro tiver um valor comum ou esperado.
- Garanta que os parâmetros obrigatórios venham antes dos opcionais na definição da função.
- Combine valores padrão com lógica interna apenas quando necessário.

Em primeiro lugar, é preciso lembrar que existem **três maneiras de declarar variáveis** em Javascript:

```
● ● ●  
1 var variável = "Sou uma variável"  
2  
3 let variávelLet = "Sou do tipo let"  
4  
5 const constante= "Sou uma constante"
```

Definimos o tipo de variável e definimos o valor que ela recebe, nesse caso uma STRING.



Quando declaradas na raiz do arquivo, as três formas estão corretas e funcionam da mesma maneira. Mas sabemos que a variável do **tipo const** não pode ter o seu valor atribuído alterado.

Analisando o código abaixo, conseguimos entender perfeitamente o que eles estão realizando, correto?

Atribuímos um valor a uma variável.

Temos uma função que tem como tarefa imprimir uma mensagem na tela.

E logo em seguida chamamos a função para executá-la.

```
js aula3_5.js > ...  
1 const primeiraVariavelDoCodigo = 'texto inicial';  
2  
3 function printToConsole() {  
4 | console.log(primeiraVariavelDoCodigo);  
5 }  
6  
7 printToConsole();
```

Continuando nosso exemplo, sabemos que não podemos alterar o valor atribuído a uma variável do tipo const, ao tentar alterá-la geraríamos o seguinte erro:

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2 primeiraVariavelDoCodigo = 'segundo texto';
3
4 function printToConsole() {
5   console.log(primeiraVariavelDoCodigo);
6 }
7
8 printToConsole();
```

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2 const primeiraVariavelDoCodigo = 'segundo texto';
3
4 function printToConsole() {
5   console.log(primeiraVariavelDoCodigo);
6 }
7
8 printToConsole();
```

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
primeiraVariavelDoCodigo = 'segundo texto';

TypeError: Assignment to constant variable.
    at Object. (/home/milene/javascript_hashtag/script.js:2:26)
    at Module._compile (internal/modules/cjs/loader.js:1085:14)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1114:10)
    at Module.load (internal/modules/cjs/loader.js:950:32)
    at Function.Module._load (internal/modules/cjs/loader.js:790:12)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:75:12)
    at internal/main/run_main_module.js:17:47
```

Alterar o tipo da variável para let, seria uma solução para corrigirmos esse erro.

Mas e se a variável, fosse redeclarada dentro da nossa função printToConsole? Esse erro ocorreria?

Agora que relembramos alguns conceitos sobre as variáveis, podemos entender algo muito comum entre as linguagens de programação, o conceito de **Escopos de Variáveis**.

É possível entender o escopo como '**local**' ou a '**acessibilidade**' de variáveis, funções e objetos em diferentes partes do código. Podemos definir **Escopo** também como um conjunto de regras para encontrar variáveis ou a área em que temos acesso válido a funções ou blocos.

Temos dois tipos de escopo, o **Global** e o **Local**, e vamos começar a entendê-los imaginando que o **Escopo é uma caixa!**

```
JS aula3_5.js > ...
          Escopo Global
1   const primeiraVariavelDoCodigo = 'texto inicial';
2
3   function printToConsole() { Escopo Local
4     console.log(primeiraVariavelDoCodigo);
5   }
6
7   printToConsole();
```

Ao iniciarmos um código, o Javascript cria o **ESCOPO GLOBAL**, uma caixa que envolve todo o nosso código. Dentro dessa caixa iremos criar outras caixas.

Cada caixa irá guardar suas funções, variáveis e objetos.

Todas as variáveis declaradas neste espaço podem ser **acessadas a qualquer momento e em qualquer lugar** do seu código.

Um exemplo de **ESCOPO LOCAL** é criado sempre que definimos uma função.

Ele é um **escopo mais restrito**, pois qualquer variável, função e parâmetros declarados dentro dela serão **acessíveis apenas dentro dela**, mas não fora dela.



ESCOPO GLOBAL

O **Escopo Global** é o escopo mais amplo em um programa Javascript, e inclui todas as variáveis e funções que são definidas fora de qualquer função ou bloco de código.

Repare no exemplo abaixo, mesmo sem passar nada como parâmetro para a função printToConsole, ela consegue acessar a variável **primeiraVariavelDoCodigo**, pois foi declarada no **Escopo Global**, ou seja, as variáveis que estão no **Escopo Global** podem ser acessadas dentro das funções.

```
1 //Escopo Global
2 const primeiraVariavelDoCodigo = 'texto inicial';
3
4 function printToConsole() {
5   console.log(primeiraVariavelDoCodigo); // primeiraVariavelDoCodigo é GLOBAL, então é acessível aqui.
6 }
7
8 printToConsole();
9
```

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
texto inicial
```



ESCOPO LOCAL

O **Escopo local** permite que a variável tenha sua **abrangência limitada**, o mais comum é ser dentro de uma função, ou seja, ela só é visível dentro daquela função e não se confunde com outras partes do código do seu programa.

Então repare que no caso a seguir, declaramos novamente a variável **const primeiraVariavelDoCodigo** com um novo valor dentro da função, e como está sendo redeclarada dentro do escopo local da função, o seu retorno será o novo valor atribuído.

```
JS aula3_5.js > ...
1  const primeiraVariavelDoCodigo = 'texto inicial';
2
3  function printToConsole() { Escopo Local da
4    const primeiraVariavelDoCodigo = 'segundo texto';
5    console.log(primeiraVariavelDoCodigo);
6  }
7
8  printToConsole();
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
segundo texto
```

Ou seja, se entendermos que o **conceito de Escopo** nos permite, em determinadas partes do código, ter acesso a determinadas variáveis, a questão levantada sobre o redeclararmos a variável **primeiraVariavelDoCodigo** poder ter seu valor alterado se torna **verdadeiro dentro do Escopo local da função printToConsole**, sem apresentar nenhum erro.

Mas se tentarmos alterar novamente o valor da variável, só que nesse momento dentro do escopo da função, o mesmo erro irá ocorrer, pois não podemos reatribuir valores para variáveis do tipo const que estão localizadas no mesmo escopo.

```
JS aula3_5.js > printToConsole > primeiraVariavelDoCodigo
1  const primeiraVariavelDoCodigo = 'texto inicial';
2
3  function printToConsole() {
4      const primeiraVariavelDoCodigo = 'segundo texto';
5      const primeiraVariavelDoCodigo = 'terceiro texto';
6      console.log(primeiraVariavelDoCodigo);
7  }
8
9  printToConsole();
```

Para continuarmos com alguns exemplos precisamos saber que é possível declarar funções dentro de outras funções. Ela é declarada com a sintaxe de uma função só que dentro de outra função.

Esse tipo de declaração chama-se Função aninhada e inicialmente é acessível apenas ao Escopo que a contém, mas sua referência pode ser retornada para outro escopo (Mas não se preocupe nesse momento em entender a importância desse contexto. Exploraremos mais sobre isso ao longo do curso).

```
JS aula3_5.js > printToConsole      Escopo Global
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() { Escopo Local função PrintToConsole
4     const primeiraVariavelDoCodigo = 'segundo texto';
5     function secondFunction() { Escopo Local função secondFunction
6         console.log('imprimindo segunda mensagem');
7     }
8     console.log(primeiraVariavelDoCodigo);
9     secondFunction();
10 }
11
12 printToConsole();
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
segundo texto
imprimindo segunda mensagem
```

Uma função declarada dentro de outra função, apenas irá viver durante o escopo da função pai, ou seja, a **função secondFunction** apenas existe dentro do escopo da **função printToConsole**.

Assim no exemplo ao lado, estamos definindo que a função **secondFunction** irá retornar e invocamos ela dentro do escopo da função **printToConsole**.

Dessa forma, o retorno desse código será a variável declarada dentro do **escopo printToConsole** e o que foi definido dentro do **console.log()** da função **secondFunction**.

A ideia de escopo é super importante, é um conceito que permite **controlar melhor o fluxo** do nosso código.

Á medida que você cria funções dentro de funções, entendemos que estamos colocando uma caixa dentro da outra.

Porém a caixa maior nunca terá acesso as informações da caixa menor, pois é como se elas estivesse fechadas.

Analisando o exemplo ao lado, podemos verificar:

Primeiro estamos declarando uma segunda variável dentro do escopo da função.

E tentamos imprimir seu valor no escopo global do código.

Esse código nos retorna um erro, que mostra a variável não está definida, ou seja, ele não consegue encontrar o valor que atribuímos a ela.

O **erro** ocorre pois estamos tentando capturar uma informação dentro de um **Escopo Local (caixa menor)**, na qual nosso **Escopo Global (caixa maior)** não consegue ter acesso.

Diferente dos exemplos anteriores, onde nossa função tem acesso ás informações do Escopo Global do nosso código, o Global **NUNCA** terá acesso aos valores contidos nos Escopos de Variáveis das funções ou blocos (Local)

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() {
4     const outraVariavel = 'Texto dentro do escopo local';
5     console.log(outraVariavel); Escopo Local
6 }
7
8 // printToConsole();
9
10 console.log(outraVariavel); Escopo Global
```

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
console.log(outraVariavel);
^

ReferenceError: outraVariavel is not defined
at Object. (/nome/mídia/javascript_nasntag/script.js:10:13)
at Module._compile (internal/modules/cjs/loader.js:1085:14)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:1114:10)
at Module.load (internal/modules/cjs/loader.js:950:32)
at Function.Module.load (internal/modules/cjs/loader.js:790:12)
at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:75:12)
at internal/main/run_main_module.js:17:47
```

Outro conceito importante para entendermos sobre acessos das variáveis é a **CADEIA DE ESCOPO**.

Uma **CADEIA DE ESCOPO** é aplicada de dentro para fora, ou seja, dentro do escopo da função iremos procurar a variável, caso não encontremos a variável declarada, vamos procurar no próximo escopo **EXTERNO**, aquele que está envolvendo nossa função, podendo ser outra função.

Esse processo irá continuar até atingir o Escopo Global, se a variável existir será usada, senão ocorrerá um erro.
Para compreender o conceito da Cadeia de Escopo, vamos utilizar o exemplo abaixo:

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() {
4     const primeiraVariavelDoCodigo = 'segundo texto';
5     function secondFunction() {
6         console.log(primeiraVariavelDoCodigo);
7     }
8     console.log(primeiraVariavelDoCodigo);
9     secondFunction();
10 }
11
12 printToConsole();
13
```

Como podemos ver, criamos duas variáveis: No escopo global e outra no escopo da **função printToConsole**.

Declaramos uma segunda função dentro da função printToConsole.

A **função secondFuntion** deve imprimir a variável primeiraVariavelDoCodigo, no entanto, não declaramos essa variável dentro do seu escopo.

Então será o retorno desse programa?

É nesse momento que o conceito da **Cadeia de Escopo** será aplicado.

A **função secondFunction** irá procurar a variável dentro de seu escopo local, como ela não possui a sua declaração, ela irá para o escopo externo mais próximo, o escopo da **função printToConsole** e irá aplicar o valor que está sendo definido para a **variável primeiraVariavelDoCodigo** e o retorno do programa será:

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() {
4     const primeiraVariavelDoCodigo = 'segundo texto';
5     function secondFunction() {
6         console.log(primeiraVariavelDoCodigo);
7     }
8     console.log(primeiraVariavelDoCodigo);
9     secondFunction();
10}
11
12 printToConsole();
13
```

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
segundo texto
segundo texto
```

Módulo 4 – Escopos de Variáveis (11 /16)

Agora que conseguimos identificar os tipos de Escopos de Variáveis e seus retornos, vamos analisar passo a passo os próximos trechos de código:

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() {
4   const primeiraVariavelDoCodigo = 'segundo texto';
5   function secondFunction() {
6     const primeiraVariavelDoCodigo = 'terceiro texto';
7     console.log(primeiraVariavelDoCodigo);
8   }
9   console.log(primeiraVariavelDoCodigo);
10  secondFunction();
11 }
12
13 printToConsole();
```

Exemplo 1

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
segundo texto
terceiro texto
```

```
1 const primeiraVariavelDoCodigo = 'texto inicial';
2
3 function printToConsole() {
4   // const primeiraVariavelDoCodigo = 'segundo texto';
5   function secondFunction() {
6     const primeiraVariavelDoCodigo = 'terceiro texto';
7     console.log(primeiraVariavelDoCodigo);
8   }
9   console.log(primeiraVariavelDoCodigo);
10  secondFunction();
11 }
12
13 printToConsole();
```

Exemplo 2

```
C:\Users\usuario\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
texto inicial
terceiro texto
```

Passo 1 : Vamos verificar onde estão sendo executadas as instruções de imprimir na tela.

-> **os console.logs estão nas Linhas 7 e 9.**

Passo 2 : A função printToConsole está sendo chamada primeiro.

-> **Linha 13.**

Passo 3 : Dentro do escopo da função printToConsole temos a declaração da variável, depois a definição da função secondFunction (Lembrando que definição não é execução).

Passo 4 : Chamamos o primeiro console.log (Linha 9), ou seja, executamos essa tarefa.

-> Para executá-la precisamos analisar se o valor desta variável está definida no seu escopo, no seu espaço, se estiver, use-a (Como é o caso do exemplo 1)

Caso não esteja sendo declarada nesse escopo (Como é o caso do exemplo 2), observe o escopo acima ou externo e verifique se a variável está sendo declarada (neste caso o escopo global).

Passo 5 : E depois execute a função secondFunction, que também possui uma variável declarada e um console log para ser executado.

-> **Linha 7.**



Resumindo - O Que É Escopo?

O escopo é o contexto onde as variáveis e funções são declaradas e acessadas. Ele controla a **visibilidade** e a **vida útil** das variáveis. Existem três tipos principais de escopo em JavaScript:

- **Escopo Global**

Variáveis declaradas fora de qualquer função ou bloco têm escopo global e podem ser acessadas de qualquer lugar do código.

- **Escopo Local (de Função)**

Variáveis declaradas dentro de uma função só estão disponíveis dentro daquela função.

- **Escopo de Bloco**

Introduzido com let e const, variáveis declaradas dentro de blocos {} (como em loops ou condicionais) só existem dentro do bloco.

Exemplo 1: Escopo Global

```
let global = "Sou do escopo Global";

function mensagem() {
    console.log(global); // Acessando uma variável global dentro da função
}

mensagem(); // Saída: "Sou do escopo Global"
console.log(global); // Saída: "Sou do escopo Global"
```



No código anterior:

- A variável global é declarada fora de qualquer função ou bloco, tornando-a global.
- Dentro da função mensagem, conseguimos acessar global porque ela está no escopo global.
- Fora da função, também podemos acessar global.

Exemplo 2: Escopo Local (Função)

```
function mensagem() {  
    let local = "Sou local da função"; // Variável com escopo local  
    console.log(local); // Saída: "Sou local da função"  
}  
  
mensagem();  
console.log(global); // Saída: "Sou do escopo Global"  
// console.log(local); // Gera um erro porque 'local' não está no escopo global
```

- A variável local é declarada **dentro** da função mensagem, portanto, ela só pode ser acessada dentro dessa função.
- Tentar acessar local fora da função gera um erro, pois ela está no escopo local da função.

Exemplo 3: Escopo de Bloco

```
if (true) {  
    let bloco = "Sou do bloco do IF"; // Variável com escopo de bloco  
    console.log(bloco); // Saída: "Sou do bloco do IF"  
}  
  
// console.log(bloco); // Gera um erro porque 'bloco' não está no escopo global
```

- A variável bloco é declarada dentro do bloco {} da instrução if.
- Ela só pode ser acessada **dentro** desse bloco. Fora do bloco, o JavaScript não reconhece bloco.

Diferenças Entre Escopo Global, Local e de Bloco

Tipo de Escopo	Onde é Declarado	Onde Pode Ser Acessado	Exemplo
Global	Fora de funções ou blocos	Em qualquer lugar no código	<code>let global = "Global";</code>
Local (Função)	Dentro de uma função	Apenas dentro da função onde foi declarado	<code>function mensagem() { let local; }</code>
De Bloco	Dentro de blocos {}	Apenas dentro do bloco onde foi declarado	<code>if (true) { let bloco = "Bloco"; }</code>

Boas Práticas de Escopo

- **Evite Variáveis Globais Sempre Que Possível**

Variáveis globais podem ser modificadas accidentalmente em qualquer parte do código, o que aumenta o risco de erros.

- **Prefira Escopo de Bloco com let e const**

Antes do ES6, todas as variáveis tinham escopo global ou de função. Com let e const, você pode limitar o escopo ao bloco onde a variável é necessária, tornando o código mais seguro e legível.

- **Declare Variáveis o Mais Próximo Possível de Onde Elas São Usadas**

Isso reduz a chance de confusão sobre onde a variável pode ser acessada.

Conclusão

O escopo é uma parte crucial da programação em JavaScript. Entender as diferenças entre escopo global, local e de bloco ajuda você a organizar melhor o código e evitar erros difíceis de depurar.

Dica Prática: Sempre que declarar uma variável, pergunte-se: **Onde eu preciso acessá-la?** Use escopo global apenas quando necessário, prefira escopos mais restritos para melhorar a segurança e legibilidade do código.

Módulo 5

MÉTODOS

MÉTODOS

MÉTODOS



Bem-vindo ao módulo sobre **métodos em JavaScript!** Nesta etapa do seu aprendizado, você vai explorar um dos conceitos mais poderosos e amplamente utilizados na linguagem: os métodos. Entender como eles funcionam e como utilizá-los de forma eficiente é essencial para dominar o JavaScript e desenvolver aplicações mais dinâmicas e organizadas.

O Que Vamos Aprender?

Neste módulo, abordaremos os principais aspectos relacionados a métodos, incluindo:

- **Métodos de Objetos**

Como criar e utilizar métodos personalizados em objetos.

- **Métodos Nativos**

Uma introdução aos métodos integrados do JavaScript, como os que existem para strings, arrays e objetos.

- **Contexto e this**

Vamos entender como o contexto influencia os métodos e a importância da palavra-chave `this` ao trabalhar com objetos.

- **Encadeamento de Métodos (Method Chaining)**

Como combinar múltiplos métodos em uma única chamada para manipular dados de forma eficiente.

Por Que Métodos São Importantes?

Os métodos são fundamentais porque permitem:

- **Organizar o Código:**

Ao associar funções diretamente a objetos, você pode manter ações relacionadas agrupadas e mais fáceis de localizar.

- **Reutilizar Lógica:**

Os métodos ajudam a encapsular comportamentos, facilitando sua reutilização em diferentes partes do programa.

- **Manipular Dados Com Eficiência:**

Métodos integrados, como os de strings e arrays, tornam tarefas complexas simples e rápidas de implementar.



O que são métodos?

Métodos são funções associadas a objetos ou tipos de dados em JavaScript, que permitem realizar ações específicas diretamente nesses valores. Eles são ferramentas que facilitam o trabalho com strings, arrays, números, e outros tipos de dados.

Por que usar métodos?

Eles economizam tempo e tornam o código mais legível, pois encapsulam funcionalidades comuns em comandos simples e reutilizáveis.

Exemplos de métodos nativos

Aqui estão alguns exemplos com uma breve descrição:

- **toUpperCase()**

Converte todas as letras de uma string para maiúsculas.

Exemplo prático: Transformar "javascript" em "JAVASCRIPT".

- **includes()**

Verifica se uma string ou array contém um valor específico.

Exemplo prático: Checar se a palavra "JS" está presente em "Aprender JS".

- **push()**

Adiciona um ou mais elementos ao final de um array.

Exemplo prático: Inserir valores novos em uma lista.

Onde aprender mais sobre métodos?

Além da prática, a **W3Schools** é uma excelente plataforma para aprender sobre métodos de JavaScript. Ela oferece explicações simples, exemplos e um ambiente interativo para testar o código.

- Acesse: [W3Schools JavaScript Methods](#)

Por exemplo:

- Para `toUpperCase`, veja a página: [toUpperCase no W3Schools](#)
- Para `push`, veja: [push no W3Schools](#)

The screenshot shows the W3Schools website interface. The top navigation bar includes links for Tutorials, Exercises, Certificates, Services, a search bar, and various programming language tabs like SQL, PYTHON, JAVA, PHP, HOW TO, W3.CSS, C, C++, C#, BOOTSTRAP, REACT, MYSQL, JQUERY, EXCEL, and XM. Below the navigation is a sidebar with sections for JS Reference (JS by Category, JS by Alphabet) and JavaScript (JS Arrays, JS Objects, JS Functions, etc.). The main content area displays the title "JavaScript Array Methods and Properties". A table lists several array methods with their descriptions:

Name	Description
<code>[]</code>	Creates a new Array
<code>new Array()</code>	Creates a new Array
<code>at()</code>	Returns an indexed element of an array
<code>concat()</code>	Joins arrays and returns an array with the joined arrays
<code>constructor</code>	Returns the function that created the Array prototype

Nesta aula, vamos explorar as principais diferenças entre **métodos** e **funções** em JavaScript, dois conceitos fundamentais na linguagem. Embora ambos envolvam a execução de código, eles têm papéis e contextos diferentes.

O que é uma Função?

Uma **função** é um bloco de código que pode ser chamado e executado para realizar uma tarefa específica. Ela pode receber parâmetros e retornar um valor, tornando-se reutilizável em diferentes partes do código. Funções são definidas de forma independente e podem ser usadas em qualquer lugar.

Exemplo de função:

```
function saudacao(nome) {  
    return `Olá, ${nome}`;  
}  
  
console.log(saudacao("Ana")); // Saída: "Olá, Ana"  
console.log(saudacao("Paulo")); // Saída: "Olá, Paulo"  
console.log(saudacao("José")); // Saída: "Olá, José"
```

Neste exemplo, a função saudacao recebe o parâmetro nome e retorna uma string personalizada. A função é chamada diretamente, passando diferentes valores como argumento.

Características de uma função:

- Independente: Pode ser definida fora de qualquer objeto ou contexto.
- Reutilizável: Pode ser chamada várias vezes, com diferentes valores de parâmetros.

O que é um Método?

Um **método** é uma função associada a um **objeto**. Ou seja, um método é uma função que pertence a um objeto e, normalmente, é usado para realizar uma ação sobre esse objeto. Métodos são chamados através da referência ao objeto e podem acessar e modificar as propriedades desse objeto.

Exemplo de método:

```
const pessoa = {
    nome: "Ana",
    saudacao: function () {
        return `Olá, ${pessoa.nome}`;
    },
};

console.log(pessoa.saudacao()); // Saída: "Olá, Ana"
```

Aqui, o método **saudacao** está associado ao objeto **pessoa**. Para chamar o método, usamos a notação de ponto **pessoa.saudacao()**, o que indica que estamos acessando o método do objeto.

Características de um método:

- **Associado a um objeto:** Ele é uma função que pertence a um objeto.
- **Pode acessar propriedades do objeto:** Dentro do método, você pode acessar e manipular as propriedades e outros métodos do próprio objeto.

Diferenças Principais entre Função e Método

Característica	Função	Método
Definição	Um bloco de código independente.	Uma função associada a um objeto.
Forma de Chamada	Chamado diretamente pelo nome.	Chamado através de um objeto.
Exemplo	<code>saudacao("Ana")</code>	<code>pessoa.saudacao()</code>
Contexto	Não possui vínculo com um objeto.	Está vinculado a um objeto.
Acesso ao objeto	Não tem acesso direto ao objeto.	Pode acessar as propriedades e métodos do objeto.

Resumo:

- **Funções** são blocos de código independentes, reutilizáveis, que podem ser chamados com diferentes argumentos para realizar tarefas específicas.
- **Métodos** são funções que pertencem a objetos e são usadas para operar sobre os dados desse objeto, podendo acessar suas propriedades e outros métodos.

Nesta aula, vamos aprender sobre os **métodos nativos de String** em JavaScript, que são funções pré-definidas para manipular e trabalhar com sequências de caracteres (strings). Esses métodos permitem que você modifique, busque, ou extraia partes de uma string de forma fácil e eficiente.

O que é uma String?

Em JavaScript, **String** é um tipo de dado que representa uma sequência de caracteres, como palavras, frases ou até mesmo números representados como texto. Uma string pode ser criada utilizando aspas simples, duplas ou crase (para template literals).

```
let texto = "Olá, Mundo!"
```

Métodos Nativos de String

JavaScript oferece uma série de métodos nativos para trabalhar com strings. Vamos ver alguns deles com exemplos práticos.

1. **toUpperCase()**

O método **toUpperCase()** converte todos os caracteres de uma string para **letras maiúsculas**.

```
let string = "Olá, Mundo!";
console.log(string.toUpperCase()); // Saída: "OLÁ, MUNDO!"
```



2. **toLowerCase()**

O método **toLowerCase()** converte todos os caracteres de uma string para **letras minúsculas**.

```
console.log(string.toLowerCase()); // Saída: "olá, mundo!"
```

3. **slice(start, end)**

O método **slice()** é utilizado para **extrair uma parte da string**. Ele recebe dois parâmetros: start (o índice de início) e end (o índice de término, que não será incluído). Caso o índice end não seja fornecido, a extração vai até o final da string.

```
console.log(string.slice(0, 5)); // Saída: "olá, "
console.log(string.slice(-6)); // Saída: "Mundo!" (contagem de trás para frente)
```

- **slice(0, 5)** extrai os caracteres da posição 0 até a posição 5 (não incluindo o índice 5).
- **slice(-6)** começa a contar de trás para frente, extraíndo os últimos 6 caracteres.

4. **substring(start, end)**

O método **substring()** também é usado para extrair uma parte da string. A diferença principal é que ele **não aceita índices negativos** e, caso o parâmetro start seja maior que o end, ele inverte os valores, ou seja, trata os índices como se fossem trocados.

```
console.log(string.substring(0, 5)); // Saída: "Olá, "
console.log(string.substring(7, 3)); // Saída: "Mundo"
```

- **substring(0, 5)** é semelhante ao slice(0, 5), extraíndo os primeiros 5 caracteres.
- **substring(7, 3)** inverte os parâmetros e, como resultado, extraí de 3 até 7, produzindo "Mundo".

Diferença entre slice() e substring()

- **slice()** pode usar índices negativos, o que facilita acessar partes da string começando do final. Já o **substring()** não aceita índices negativos e sempre considera o valor menor como o índice de início.
- **substring()** inverte os parâmetros start e end se o start for maior que o end. Isso não

Na aula anterior, aprendemos sobre **métodos de string**, como **toUpperCase()** para converter uma string para maiúsculas, **toLowerCase()** para converter para minúsculas, **slice()** e **substring()** para extrair partes de uma string. Esses métodos são fundamentais para manipularmos e trabalharmos com strings em JavaScript.

Agora, vamos explorar mais alguns **métodos de string** que são frequentemente usados, como **replace()** e **indexOf()**, e entender como funcionam.

1. **replace(search, replaceWith)**

O método **replace()** permite substituir parte do conteúdo de uma string por um novo valor. Ele recebe dois parâmetros:

- **search**: O valor que queremos substituir.
- **replaceWith**: O valor que irá substituir o que foi encontrado.

É importante notar que **replace() não altera a string original**, mas retorna uma nova string com a substituição.

```
let string = "Olá, Mundo!";
console.log(string.replace("o", "Javascript")); // Saída: "OlJavascript, Mundo!"
```

Neste exemplo, a letra "o" foi substituída por "**Javascript**". No entanto, apenas a primeira ocorrência de "o" foi substituída, pois **replace()** substitui a primeira ocorrência encontrada, e não todas as ocorrências.

Se quisermos substituir todas as ocorrências de um valor, podemos usar uma **expressão regular** com a flag g (global):

```
console.log(string.replace(/o/g, "Javascript")); // Saída: "Javascripta, MundJavascript!"
```

2. indexOf(search)

O método **indexOf()** retorna o **índice da primeira ocorrência** de um valor dentro de uma string. Caso o valor não seja encontrado, ele retorna **-1**.

```
let string = "Olá, Mundo!";
console.log(string.indexOf("o")); // Saída: 1
```

Aqui, o método **indexOf()** retorna 1, que é o índice da primeira ocorrência da letra "o" na string.

Caso o valor não exista:

```
console.log(string.indexOf("x")); // Saída: -1
```

Se a string não contiver o valor procurado, o **indexOf()** retorna -1.

- **replace()** é usado para substituir parte de uma string por outro valor. Ele retorna uma nova string, sem alterar a original.
- **indexOf()** retorna o índice da primeira ocorrência de um valor dentro de uma string. Caso o valor não seja encontrado, retorna -1.

Esses métodos são úteis quando precisamos fazer substituições ou procurar informações dentro de uma string.

1. Método **split()**

O **split()** é um método utilizado para dividir uma string em **um array de substrings**, com base em um delimitador que você define. Esse delimitador pode ser um caractere, uma expressão regular ou qualquer outro critério que você escolher.

Sintaxe:

```
string.split(delimitador, [limite]);
```

- **delimitador**: O valor ou padrão de separação. Pode ser uma string ou uma expressão regular.
- **limite (opcional)**: Um número que especifica o número máximo de substrings que o array pode conter.

Exemplo básico:

Vamos começar com um exemplo simples para entender como o **split()** funciona.

```
let texto = "maçã,banana,laranja,uvas";
let frutas = texto.split(","); // Aqui, usamos a vírgula como delimitador.
console.log(frutas); // Saída: ["maçã", "banana", "laranja", "uvas"]
```

Aqui, o **split(",")** divide a string texto em várias partes, usando a vírgula como delimitador. Como resultado, obtemos um array com quatro elementos.

Usando o limite de substrings:

Podemos também limitar o número de elementos no array retornado, passando um segundo parâmetro (o número máximo de divisões).

```
let frutasLimite = texto.split(", ", 3); // Limita a divisão a 3 elementos
console.log(frutasLimite); // Saída: ["maçã", "banana", "laranja"]
```

Neste exemplo, o **split()** divide a string em 3 partes, mesmo que existam mais separadores na string original.

Usando uma expressão regular como delimitador:

Você também pode usar uma **expressão regular** como delimitador para dividir a string.

```
let textoEspacos = "maçã banana laranja uvas";
let frutasEspacos = textoEspacos.split(/\s+/); // Aqui, usamos uma expressão regular
console.log(frutasEspacos); // Saída: ["maçã", "banana", "laranja", "uvas"]
```

Neste exemplo, a **expressão regular \s+** funciona como delimitador para dividir a string onde houver um ou mais espaços.

2. Método splice()

O método **splice()** é utilizado para **modificar um array**, permitindo que você remova ou adicione elementos em qualquer posição.

Sintaxe:

```
array.splice(início, quantidade, item1, item2, ...);
```

- **início**: O índice onde a operação começará.
- **quantidade**: O número de elementos a serem removidos (opcional).
- **item1, item2, ... (opcional)**: Elementos a serem adicionados no array, a partir do índice inicial.

Exemplo de remoção:

```
let frutas = ["maçã", "banana", "laranja", "uvas"];
frutas.splice(1, 2); // Remove 2 elementos a partir do índice 1.
console.log(frutas); // Saída: ["maçã", "uvas"]
```

Neste exemplo, **splice(1, 2)** começa a operação no índice 1 (o elemento "banana") e remove 2 elementos a partir dali, que são "banana" e "laranja". O array resultante contém apenas "maçã" e "uvas".

Exemplo de adição de elementos:

```
frutas.splice(1, 0, "abacaxi", "manga"); // Adiciona "abacaxi" e "manga" no índice 1  
console.log(frutas); // Saída: ["maçã", "abacaxi", "manga", "uvas"]
```

Aqui, **splice(1, 0, "abacaxi", "manga")** não remove nenhum elemento (por isso, o segundo parâmetro é 0), mas adiciona "abacaxi" e "manga" no índice 1, movendo os outros elementos para a direita.

- O **método split()** é utilizado para dividir uma string em um array, usando um delimitador específico. Você pode também limitar o número de divisões ou usar expressões regulares.
- O **método splice()** é usado para alterar o conteúdo de um array, podendo adicionar, remover ou substituir elementos em qualquer posição.

Esses métodos são essenciais para manipulação de strings e arrays em JavaScript, ajudando a realizar diversas operações de forma prática e eficiente.

Nesta aula, vamos aprender sobre alguns dos métodos nativos mais comuns para manipulação de **arrays** em JavaScript. Arrays são estruturas de dados que armazenam uma lista de elementos, e os métodos nativos nos ajudam a adicionar, remover e acessar esses elementos de forma eficiente.

O que são Métodos Nativos de Array?

Métodos nativos de array são funções predefinidas em JavaScript que permitem interagir diretamente com arrays de maneira prática e eficiente. Estes métodos são muito úteis para modificar ou acessar os elementos de um array de maneira rápida e sem a necessidade de criar código complexo.

Hoje vamos aprender sobre os seguintes métodos:

- **push()**
- **unshift()**
- **pop()**
- **shift()**



1. Método push()

O método **push()** adiciona um ou mais elementos **ao final** de um array. Ele **retorna o novo comprimento** do array após a adição dos elementos.

```
let produtos = ["Carrinho", "Boneca", "Bola"];  
  
produtos.push(50, 10, 30); // Adiciona os números ao final da lista  
console.log(produtos); // Saída: ["Carrinho", "Boneca", "Bola", 50, 10, 30]
```

Aqui, o **push()** adiciona os números **50, 10 e 30** ao final da lista de produtos.

2. Método unshift()

O método **unshift()** adiciona um ou mais elementos **no início** de um array. Assim como o **push()**, ele também **retorna o novo comprimento** do array.

```
produtos.unshift("Video Game"); // Adiciona "Video Game" no início da lista  
console.log(produtos); // Saída: ["Video Game", "Carrinho", "Boneca", "Bola", 50, 10, 30]
```

No exemplo, **unshift("Video Game")** adiciona **"Video Game"** no início do array de produtos, empurrando os outros itens para a direita.



3. Método **pop()**

O método **pop()** remove o **último elemento** de um array e o retorna. Este método altera o array original.

```
produtos.pop(); // Remove o último elemento da lista  
console.log(produtos); // Saída: ["Video Game", "Carrinho", "Boneca", "Bola", 50, 10]
```

Aqui, o **pop()** remove o último item do array, que é **30**, e retorna esse valor. O array fica com os elementos restantes.

4. Método **shift()**

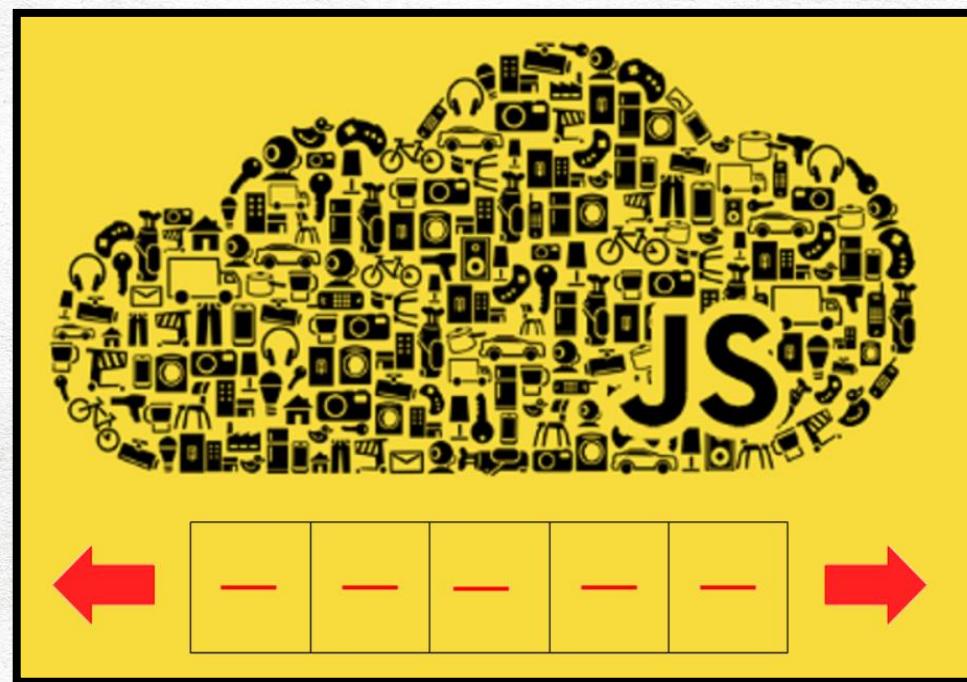
O método **shift()** remove o **primeiro elemento** de um array e o retorna, alterando o array original.

```
produtos.shift(); // Remove o primeiro elemento da lista  
console.log(produtos); // Saída: ["Carrinho", "Boneca", "Bola", 50, 10]
```

Aqui, o **shift()** remove o primeiro elemento, que é "**Video Game**", e retorna esse valor. O array agora começa com "**Carrinho**".

- **push()**: Adiciona um ou mais elementos ao final do array.
- **unshift()**: Adiciona um ou mais elementos no início do array.
- **pop()**: Remove o último elemento do array e o retorna.
- **shift()**: Remove o primeiro elemento do array e o retorna.

Esses métodos são essenciais quando precisamos modificar um array, seja adicionando novos elementos ou removendo os existentes. Eles ajudam a manter o código limpo e eficiente para manipular dados dentro de arrays.



Nesta aula, vamos aprender sobre métodos adicionais que ajudam na manipulação de arrays, especialmente quando precisamos **ordenar**, **reverter** ou **mesclar** arrays em JavaScript. Vamos aprender os seguintes métodos:

- **sort()**
- **reverse()**
- **concat()**

Esses métodos são muito úteis quando queremos reorganizar os elementos dentro de um array ou combinar arrays de maneira eficiente.

1. Método **sort()**

O método **sort()** é utilizado para ordenar os elementos de um array. Por padrão, ele ordena os **elementos como strings**, o que significa que ele pode não ordenar números da maneira que você espera. Vamos ver dois exemplos: um com letras e outro com números.

Exemplo com Letras:

```
let letras = ["d", "g", "a", "h", "b", "f", "c", "e"];
letras.sort(); // Ordena as Letras em ordem alfabética
console.log(letras); // Saída: ["a", "b", "c", "d", "e", "f", "g", "h"]
```

Exemplo com Números:

```
let numeros = [1, 6, 5, 4, 10, 8, 20, 19, 3];
numeros.sort(); // Ordena os números de forma lexicográfica (como strings)
console.log(numeros); // Saída: [1, 10, 19, 20, 3, 4, 5, 6, 8]
```

Observe que o **sort()** pode não ordenar os números corretamente, porque ele os classifica como strings, ou seja, ele faz a comparação lexicográfica (alfabética). Para ordenar corretamente números, precisamos usar uma função de comparação personalizada.

Ordenando Números Corretamente:

```
numeros.sort((a, b) => a - b); // Ordenação numérica
console.log(numeros); // Saída: [1, 3, 4, 5, 6, 8, 10, 19, 20]
```

Agora, com a função **(a, b) => a - b**, garantimos que os números sejam ordenados numericamente.

2. Método **reverse()**

O método **reverse()** reverte a ordem dos elementos de um array, ou seja, coloca o último elemento na primeira posição e assim por diante.

```
letras.reverse(); // Reverte a ordem das letras  
console.log(letras); // Saída: ["h", "g", "f", "e", "d", "c", "b", "a"]
```

- **reverse()** altera o array original e inverte a posição de todos os seus elementos.

```
numeros.reverse(); // Reverte a ordem dos números  
console.log(numeros); // Saída: [20, 19, 10, 8, 6, 5, 4, 3, 1]
```

- **reverse()** também funciona com números, invertendo a ordem dos elementos no array.

3. Método concat()

O método **concat()** é utilizado para **mesclar dois ou mais arrays** em um único array. Ele não modifica os arrays originais, mas retorna um novo array contendo os elementos de todos os arrays combinados.

Exemplo de Mesclagem de Arrays:

```
let mesclaArray = letras.concat(numeros); // Mescla os arrays letras e numeros
console.log(mesclaArray); // Saída: ["h", "g", "f", "e", "d", "c", "b", "a", 20, 19, 10, 8, 6, 5, 4, 3, 1]
```

No exemplo acima, o **concat()** combina os arrays **letras** e **numeros** em um novo array. O array resultante contém todos os elementos dos dois arrays, com os elementos de **letras** seguidos pelos de **numeros**.

Resumo:

- **sort()**: Ordena os elementos de um array. Quando usado com números, é necessário passar uma função de comparação.
- **reverse()**: Reverte a ordem dos elementos em um array.
- **concat()**: Mescla dois ou mais arrays em um único array, criando um novo array.

Esses métodos são poderosos e muito usados no dia a dia de programação em JavaScript.

Nesta aula, vamos aprender sobre dois métodos importantes para manipulação de arrays: **indexOf()** e **splice()**. Esses métodos nos ajudam a localizar, remover e até substituir elementos em um array de forma eficiente.

1. Método **indexOf()**

O método **indexOf()** retorna o **índice** (posição) da **primeira ocorrência** de um valor dentro de um array. Se o valor não for encontrado, ele retorna **-1**.

```
let frutas = ["maça", "uva", "laranja"];

console.log(frutas.indexOf("laranja")); // Saída: 2
```

No exemplo, **indexOf("laranja")** procura pelo item "**laranja**" no array **frutas**. Como "**laranja**" está na posição **2** (lembre-se de que a contagem começa em 0), o método retorna **2**.

Se tentássemos buscar um item que não existe no array, como "**abacaxi**", o método retornaria **-1**:

```
console.log(frutas.indexOf("abacaxi")); // Saída: -1
```

2. Método **splice()**

O método **splice()** é bastante poderoso e permite **adicionar, remover e substituir elementos** em um array. Ele altera o array original.

Sintaxe:

```
array.splice(início, quantidade, item1, item2, ...);
```

- **início**: A posição no array onde a operação deve começar.
- **quantidade**: O número de elementos a serem removidos a partir da posição de início.
- **item1, item2, ...**: Os itens que serão adicionados no lugar dos itens removidos. (Opcional)

Vamos ver alguns exemplos:

1. Remover um Elemento:

```
console.log(frutas.splice(0, 1)); // Saída: ["maça"]
console.log(frutas); // Saída: ["uva", "laranja"]
```

No exemplo acima, **splice(0, 1)** remove o elemento na posição **0** (o primeiro elemento, "**maça**"). O método retorna o elemento removido, que foi "**maça**". O array original agora contém **["uva", "laranja"]**.

2. Substituir um Elemento:

```
console.log(frutas.splice(2, 1, 10)); // Saída: ["laranja"]
console.log(frutas); // Saída: ["uva", 10]
```

Aqui, **splice(2, 1, 10)** faz o seguinte:

- **2**: Começa na posição **2** (onde está o item "**laranja**").
- **1**: Remove **1** elemento (o item "**laranja**").
- **10**: Adiciona o número **10** no lugar de "**laranja**".

Após a execução, o item "**laranja**" é removido e substituído por **10**.

Recapitulando os Métodos

- **indexOf()**:
 - Localiza a posição de um valor no array e retorna seu **índice**.
 - Retorna **-1** se o valor não for encontrado.
- **splice()**:
 - **Remover** elementos: array.splice(início, quantidade).
 - **Substituir** elementos: array.splice(início, quantidade, novoItem1, novoItem2, ...).
 - **Adicionar** elementos: array.splice(início, 0, item1, item2, ...).

Esses métodos são essenciais para manipularmos arrays de maneira eficiente, seja para localizar elementos ou fazer alterações diretas no conteúdo de um array.



Nesta aula, vamos explorar alguns métodos úteis para verificar a presença de valores em um array e para realizar operações mais complexas, como verificar se todos ou algum elemento do array atende a uma condição específica. Também vamos aprender como combinar os elementos de um array em uma única string usando o método `join()`.

1. Método `includes()`

O método `includes()` verifica se um determinado valor está presente em um array. Ele retorna `true` se o valor for encontrado e `false` caso contrário.

```
let frutas = ["maça", "melão", "manga", "kiwi"];

console.log(frutas.includes("manga")); // Saída: true
console.log(frutas.includes(50)); // Saída: false
```

- No primeiro exemplo, `includes("manga")` retorna `true` porque o valor "`manga`" está presente no array `frutas`.
- No segundo exemplo, `includes(50)` retorna `false` porque o número `50` não está presente no array.

2. Método **every()**

O método **every()** testa se **todos os elementos** de um array atendem a uma condição especificada. Ele executa a função fornecida para cada elemento do array e retorna **true** se todos os elementos satisfizerem a condição. Caso contrário, ele retorna **false**.

```
console.log(  
    frutas.every(function (fruta) {  
        return fruta.includes("ma");  
    })  
); // Saída: true
```

No exemplo acima, a função **every()** verifica se **todas** as frutas do array contêm a substring "**ma**". A função **fruta.includes("ma")** retorna **true** para os elementos "**maça**" e "**melão**", mas como "**manga**" e "**kiwi**" também são testados, e "**kiwi**" não contém "**ma**", o método retorna **false**.

3. Método **some()**

O método **some()** verifica se **algum elemento** de um array atende a uma condição específica. Ele retorna **true** se ao menos um elemento satisfizer a condição e **false** caso contrário.

```
console.log(  
    frutas.some(function (fruta) {  
        return fruta.includes("k");  
    })  
); // Saída: true
```

No exemplo, a função **some()** verifica se **algum** dos elementos do array contém a letra "k". O "kiwi" atende a essa condição, portanto, o método retorna **true**.

Em JavaScript, **propriedades** são características ou valores que pertencem a um objeto. Elas representam informações ou dados armazenados dentro do objeto. Essas propriedades são compostas por uma chave e um valor.

Exemplificando Propriedades dentro de um Objeto

No código abaixo, temos um objeto **objeto** com uma propriedade e um método:

```
const objeto = {  
    propriedade: 10, // Propriedade  
    saudacao: function () { // Método  
        console.log("Olá eu sou um método");  
    },  
};  
  
console.log(objeto.propriedade); // Acessando a propriedade  
objeto.saudacao(); // Chamando o método
```

- **Propriedade:** No caso do objeto, a propriedade **propriedade** tem o valor **10**.
- **Método:** O **método saudacao** é uma função que, ao ser chamada, exibe uma mensagem no console.

As **propriedades** armazenam **valores** (como números, strings, arrays, etc.), enquanto os **métodos** são funções associadas ao objeto que podem **realizar ações**.

Acessando as Propriedades

Para acessar o valor de uma propriedade em um objeto, usamos o **ponto (.)** seguido do nome da propriedade:

```
console.log(objeto.propriedade); // Saída: 10
```

Diferença entre Propriedades e Métodos

- **Propriedade:** Um valor simples armazenado dentro de um objeto. Exemplo: objeto.propriedade.
- **Método:** Uma função que é executada ao ser chamada. Exemplo: objeto.saudacao().

Propriedades Nativas

Assim como podemos criar nossas próprias propriedades dentro de objetos, tipos de dados nativos em JavaScript também têm propriedades integradas.

Exemplo de Propriedade length

- **Strings:** A propriedade **length** retorna o número de caracteres em uma string.

```
const string = "Eu sou louco por Javascript!!!";
console.log(string.length); // Saída: 29
```

Aqui, **string.length** retorna **29**, que é o número de caracteres na string "Eu sou louco por Javascript!!!".

Arrays: A propriedade **length** também é utilizada para obter o número de elementos em um array.

```
const array = [1, 30, 56, 4];
console.log(array.length); // Saída: 4
```

Neste exemplo, **array.length** retorna **4**, que é a quantidade de elementos no array.

Recapitulando

- **Propriedades** em objetos são dados ou características de um objeto, acessados através de uma chave (ex: objeto.propriedade).
- **Métodos** são funções dentro de um objeto que realizam ações (ex: objeto.saudacao()).
- **Propriedades nativas** como **length** são usadas com strings e arrays para obter o número de caracteres ou elementos respectivamente.

Com isso, você pode entender que a diferença principal é que propriedades são valores, enquanto métodos são ações, mas ambos estão ligados a objetos.

Os **objetos** em JavaScript possuem métodos nativos que permitem manipular, copiar, e modificar suas propriedades e estrutura de forma eficaz. Esses métodos são extremamente úteis para lidar com dados complexos e dinâmicos.

Conceitos e Fundamentos

- **O que são métodos nativos de objetos?**

Métodos nativos de objetos são funções predefinidas no JavaScript que nos ajudam a manipular objetos. Eles permitem:

- Copiar objetos.
- Adicionar ou alterar propriedades.
- Excluir propriedades.
- Congelar ou selar objetos, entre outros.

- **Por que utilizá-los?**

- Automatizam tarefas repetitivas.
- Garantem a integridade dos dados.
- Tornam o código mais limpo e eficiente.

- **Objetos como Estruturas Fundamentais**

Os objetos são usados para armazenar coleções de dados e entidades complexas. Trabalhar com métodos nativos facilita o gerenciamento dessas estruturas.

Nesta aula, exploraremos três métodos nativos essenciais para manipular objetos no JavaScript: **Object.keys()**, **Object.values()**, e **Object.entries()**. Esses métodos são fundamentais para acessar as propriedades e valores de objetos de forma organizada e eficiente.

1. Object.keys()

Este método retorna um array com todas as **chaves** (propriedades) de um objeto.

```
const produto = {  
    nome: "Laptop",  
    preco: 2500,  
    disponibilidade: true,  
    emEstoque: 10,  
};  
  
console.log(Object.keys(produto));  
// Saída: [ 'nome', 'preco', 'disponibilidade', 'emEstoque' ]
```

Quando usar?

- Para listar as propriedades de um objeto.

2. Object.values()

Este método retorna um array com todos os **valores** das propriedades de um objeto.

```
console.log(Object.values(produto));
// Saída: [ 'Laptop', 2500, true, 10 ]
```

Quando usar?

- Para acessar os valores de um objeto diretamente.
- Ao calcular totais ou realizar operações baseadas em valores.

3. Object.entries()

Este método retorna um array de arrays, onde cada sub-array contém um par **[chave, valor]** do objeto.

```
console.log(Object.entries(produto));
/* Saída:
[
  [ 'nome', 'Laptop' ],
  [ 'preco', 2500 ],
  [ 'disponibilidade', true ],
  [ 'emEstoque', 10 ]
]
```

Quando usar?

- Para iterar tanto pelas chaves quanto pelos valores de um objeto.
- Para transformar objetos em estruturas mais flexíveis, como mapas ou arrays.

Comparação e Casos Práticos

Método	Retorno	Uso Principal
<code>Object.keys()</code>	Array de chaves	Listar ou iterar propriedades
<code>Object.values()</code>	Array de valores	Acessar dados
<code>Object.entries()</code>	Array de pares [chave, valor]	Trabalhar com pares chave-valor

Os métodos **Object.keys()**, **Object.values()**, e **Object.entries()** são poderosos para explorar e manipular objetos. Eles fornecem formas flexíveis de acessar propriedades e valores, facilitando o trabalho com dados estruturados.

1. Object.assign()

Este método é usado para copiar valores de um ou mais objetos de origem para um objeto de destino.

Exemplo 1: Adicionar Novas Propriedades

```
const produto = {  
    nome: "Laptop",  
    preco: 2500,  
    disponibilidade: true,  
};  
  
Object.assign(produto, { emEstoque: 10, categoria: "Eletrônicos" });  
console.log(produto);  
// Saída: { nome: "Laptop", preco: 2500, disponibilidade: true, emEstoque: 10, categoria: "Eletrônicos" }
```

Exemplo 2: Combinar Vários Objetos

```
const pessoa = { nome: "Carlos", idade: 30 };
const trabalho = { profissao: "Engenheiro", cidade: "São Paulo" };
const funcionario = {};

Object.assign(funcionario, pessoa, trabalho);
console.log(funcionario);
// Saída: { nome: "Carlos", idade: 30, profissao: "Engenheiro", cidade: "São Paulo" }
```

2. Object.defineProperties()

Permite definir novas propriedades ou modificar propriedades existentes, especificando configurações detalhadas.

Exemplo: Adicionar ou Modificar uma Propriedade

```
object.defineProperty(funcionario, "salario", {  
    value: 2000,  
    enumerable: true, // Visível em loops  
    writable: true, // Pode ser modificado  
    configurable: true, // Pode ser deletado  
});  
console.log(funcionario);  
// Saída: { nome: "Carlos", idade: 30, profissao: "Engenheiro", cidade: "São Paulo", salario: 2000 }  
  
delete funcionario.salario;  
console.log(funcionario);  
// Saída: { nome: "Carlos", idade: 30, profissao: "Engenheiro", cidade: "São Paulo" }
```

3. delete

Permite remover uma propriedade de um objeto.

Exemplo: Remover uma Propriedade

```
delete funcionario.salario;  
console.log(funcionario);  
// Saída: { nome: "Carlos", idade: 30, profissao: "Engenheiro", cidade: "São Paulo" }
```

Resumo Prático

- **Object.assign(destino, ...origens)**: Copia as propriedades de um ou mais objetos de origem para um objeto destino.
- **Object.defineProperties(objeto, propriedade, configuração)**: Adiciona ou altera propriedades com configurações detalhadas.
- **delete objeto.propriedade**: Remove uma propriedade de um objeto.

Esses métodos são fundamentais para a manipulação avançada de objetos, especialmente quando trabalhamos com dados dinâmicos ou estruturas de objetos complexas.

Nesta aula, aprenderemos sobre o método **Object.create()** e suas aplicações, além de explorar propriedades herdadas e próprias de objetos em JavaScript, utilizando exemplos práticos.

O que é o Método Object.create()?

O método **Object.create()** é usado para criar um novo objeto, especificando outro objeto como o protótipo do novo. Isso permite que o novo objeto herde propriedades e métodos do protótipo.

Por que usar Object.create()?

- **Reutilização de código:** Permite criar objetos que compartilham comportamentos comuns, herdando propriedades e métodos de um protótipo.
- **Organização:** Facilita a separação entre as propriedades herdadas e aquelas adicionadas diretamente ao novo objeto.

Entendendo o Código

Definindo Protótipo e Criando Objeto

No código abaixo, criamos um objeto chamado pessoa com propriedades e métodos. Depois, criamos joao como um novo objeto que herda de pessoa.

```
const pessoa = {
  cidade: "Rio de Janeiro",
  surf: true,
  falar: function () {
    console.log("Olá");
  },
};

const joao = Object.create(pessoa);
```



No código anterior

- joao herda as propriedades cidade, surf, e o método falar de pessoa.
- Essas propriedades e métodos podem ser acessados diretamente a partir de joao, mas elas pertencem ao protótipo, e não diretamente ao objeto joao.

Exemplo de Uso do Objeto Criado

```
joao.falar(); // "Olá"  
console.log(joao.cidade, joao.surf); // "Rio de Janeiro", true
```

Adicionando Propriedades ao Objeto

Embora joao herde propriedades e métodos de pessoa, podemos adicionar propriedades específicas diretamente a ele:

```
joao.nome = "João";  
joao.idade = 30;  
console.log(joao);  
/* Saída:  
{ nome: 'João', idade: 30 }  
*/
```

Propriedades Herdadas e Próprias

hasOwnProperty()

O método **hasOwnProperty()** verifica se uma propriedade pertence diretamente ao objeto (não ao protótipo). Exemplo:

```
console.log(joao.hasOwnProperty("nome")); // true  
console.log(joao.hasOwnProperty("surf")); // false (vem do protótipo)
```

Iterando Propriedades

Podemos diferenciar propriedades herdadas das próprias ao iterar sobre um objeto:

```
for (let prop in joao) {  
    if (joao.hasOwnProperty(prop)) {  
        console.log(`Propriedade própria: ${prop}`);  
    } else {  
        console.log(`Propriedade herdada: ${prop}`);  
    }  
}  
/* Saída:  
Propriedade própria: nome  
Propriedade própria: idade  
Propriedade herdada: cidade  
Propriedade herdada: surf  
Propriedade herdada: falar  
*/
```

Definindo um Novo Protótipo

Criamos outro objeto, carro, sem relação com pessoa, mas podemos utilizá-lo como protótipo em novos objetos:

```
const carro = {  
    modelo: "Corolla",  
    marca: "Toyota",  
};  
  
const novoCarro = Object.create(carro);  
novoCarro.cor = "Prata";  
console.log(novoCarro);  
/* Saída:  
{ cor: 'Prata' }  
*/  
  
console.log(novoCarro.modelo); // "Corolla" (herdado de carro)
```

- **Object.create()**: Cria um novo objeto baseado em outro, permitindo herança de propriedades e métodos.

Propriedades Herdadas e Próprias:

- Use hasOwnProperty() para identificar se uma propriedade pertence diretamente ao objeto.
- Propriedades herdadas estão disponíveis através do protótipo.

Organização do Código:

- Object.create() é útil para criar objetos relacionados, organizando suas propriedades e métodos em protótipos.

Nesta aula, além de aprendermos sobre métodos personalizados, vamos explorar o conceito do `this`, que é essencial para entender como métodos funcionam dentro de objetos no JavaScript.

O que é `this`?

O `this` é uma palavra-chave especial no JavaScript que referencia o **contexto** no qual uma função está sendo executada.

- Dentro de um método, `this` se refere ao **objeto** ao qual o método pertence.
- O valor de `this` muda dependendo de como e onde a função é chamada.

Por que `this` é importante?

- **Acesso às Propriedades do Objeto:** Permite que métodos acessem e manipulem as propriedades do objeto ao qual pertencem.
- **Reutilização de Código:** Usar `this` torna métodos mais dinâmicos e reutilizáveis em diferentes objetos.
- **Evita Ambiguidade:** Diferencia entre variáveis locais, globais e propriedades do objeto.

O que são Métodos Personalizados?

Métodos personalizados são funções definidas dentro de um objeto que realizam operações específicas relacionadas às propriedades desse objeto.

- **Personalizados:** Criados pelo programador para atender necessidades específicas.
- **Relacionados ao Objeto:** Trabalham diretamente com os valores e estados internos do objeto, usando `this` para acessar suas propriedades.

Por que Usar Métodos Personalizados?

- **Organização:** Centraliza funcionalidades relacionadas em um único objeto.
- **Reutilização:** Torna o código mais reutilizável e modular.
- **Legibilidade:** Ajuda a entender o comportamento esperado do objeto.
- **Encapsulamento:** Esconde detalhes internos do objeto e expõe apenas os métodos necessários.

Boas Práticas ao Criar Métodos Personalizados

- **Nome Descritivo:** Escolha nomes que indicam claramente a ação do método (ex.: somar, definirValores).
- **Relacionamento ao Objeto:** Use `this` para trabalhar com as propriedades do objeto.
- **Evite Dependência Externa:** Métodos personalizados devem funcionar apenas com as propriedades e dados internos do objeto.

Entendendo o Uso de this nos Métodos Personalizados

Exemplo Prático

Vamos observar como o this é usado no objeto calculadora:

O que está acontecendo?

- `this.valor1` e `this.valor2` referem-se às propriedades do próprio objeto `calculadora`.
- Quando o método é chamado, como `calculadora.somar()`, o `this` dentro do método aponta para o objeto `calculadora`.
- Sem `this`, o método não teria como saber que precisa acessar as propriedades do objeto em que está inserido.

```
const calculadora = {
  valor1: 0,
  valor2: 0,

  definirValores: function (v1, v2) {
    this.valor1 = v1;
    this.valor2 = v2;
  },

  somar: function () {
    return this.valor1 + this.valor2; //calculadora.valor1 + calculadora.valor2;
  },

  subtrair: function () {
    return this.valor1 - this.valor2; //calculadora.valor1 - calculadora.valor2;
  },

  multiplicar: function () {
    return this.valor1 * this.valor2; //calculadora.valor1 * calculadora.valor2;
  },

  dividir: function () {
    return this.valor1 / this.valor2; //calculadora.valor1 / calculadora.valor2;
  },
};
```

O Contexto de this Pode Mudar

O valor de this depende de **como** uma função é chamada.

Exemplo: Chamada Normal

Aqui, this refere-se ao objeto objeto, pois o método foi chamado diretamente no contexto do objeto.

```
const objeto = {
  nome: "Ana",
  apresentar: function () {
    return `Meu nome é ${this.nome}`;
  },
};

console.log(objeto.apresentar()); // "Meu nome é Ana"
```

Exemplo: Perda de Contexto

Nesse caso, this não aponta mais para o objeto original, pois a função foi chamada fora do contexto do objeto.

```
const apresentarFora = objeto.apresentar;
console.log(apresentarFora()); // Erro ou undefined, pois this não aponta mais para objeto
```

Adicionando Novos Métodos

Uma das vantagens de métodos personalizados é a flexibilidade para adicionar novos comportamentos ao objeto.

Exemplo: Exponenciação

```
calculadora.exponenciacao = function () {  
    return this.valor1 ** this.valor2;  
};  
  
console.log(calculadora.exponenciacao()); // 5 elevado a 20
```

- Aqui, adicionamos o método exponenciacao ao objeto calculadora.
- Ele utiliza this para acessar valor1 e valor2 e calcula a potência.

Vantagens do Uso de Métodos Personalizados

- **Flexibilidade:** Facilmente ajustável e expansível com novos métodos.
- **Encapsulamento:** Esconde os detalhes internos de como os valores são calculados.
- **Reutilização:** O objeto pode ser reutilizado em diferentes partes do programa sem precisar repetir o código das operações.
- Métodos personalizados são funções associadas a objetos para realizar ações específicas.
- Use this para acessar propriedades internas do objeto dentro dos métodos.
- Boas práticas incluem nomes descritivos e encapsulamento de funcionalidades.

O tipo Number em JavaScript é usado para representar valores numéricos, sejam eles inteiros ou números de ponto flutuante (decimais). Além de armazenar números, o Number possui métodos nativos que ajudam na manipulação e formatação desses valores.

Conceitos Fundamentais

- **O que é o Tipo Number?**
 - Representa valores numéricos.
 - Pode ser usado para realizar cálculos matemáticos, verificar propriedades e formatar números.
- **Por que os Métodos Nativos de Number São Úteis?**
 - **Verificação:** Permitem checar características como se o número é inteiro ou finito.
 - **Formatação:** Facilitam a exibição de números com casas decimais ou precisão específicas.
 - **Precisão:** Auxiliam no controle da exibição para evitar erros de arredondamento visual.

Métodos Nativos do Tipo Number

1. Number.isInteger(value)

Verifica se o valor passado é um número inteiro.

- Retorna true se o valor for um inteiro.
- Retorna false caso contrário.

Exemplo Prático

```
let inteiro = 42;
let numeroPontoFlutuante = 3.4567;

console.log(Number.isInteger(inteiro)); // true
console.log(Number.isInteger(numeroPontoFlutuante)); // false
```

2. .toFixed(digits)

Formata um número para ter um número específico de casas decimais.

- digits: número de casas decimais (padrão é 0).
- Retorna uma string com o número formatado.

Exemplo Prático

```
let numero = 3.4567;

console.log(numero.toFixed(2)); // "3.46" (arredonda para 2 casas decimais)
console.log(numero.toFixed(0)); // "3" (sem casas decimais)
```

3. .toPrecision(digits)

Formata um número com uma precisão específica, limitando o número total de dígitos exibidos.

- digits: número total de dígitos (antes e depois do ponto decimal).
- Pode exibir números em **notação científica** quando necessário.

Exemplo Prático

```
let numero = 3.4567;

console.log(numero.toPrecision(3)); // "3.46" (3 dígitos totais)
console.log(numero.toPrecision(2)); // "3.5"
console.log(numero.toPrecision(1)); // "3"

let inteiro = 42;
console.log(inteiro.toPrecision(2)); // "42"
console.log(inteiro.toPrecision(3)); // "42.0" (3 dígitos totais, incluindo casas decimais)
```

Passo a Passo do Código

• **Number.isInteger:**

- Verifica se inteiro é um número inteiro (true).
- Verifica que numeroPontoFlutuante não é um inteiro (false).

• **.toFixed:**

- Formata numeroPontoFlutuante com 2 casas decimais: "3.46".
- Exibe o número sem casas decimais com arredondamento padrão.

• **.toPrecision:**

- Reduz ou ajusta o número exibido ao total de dígitos especificados.
- Pode exibir números em notação científica, dependendo do contexto.

```
let inteiro = 42;
let numeroPontoFlutuante = 3.4567;
let numeroPontoFlutuante2 = 3.4537;

// Verificar se o número é um inteiro - retornar booleano
console.log(Number.isInteger(inteiro)); // true
console.log(Number.isInteger(numeroPontoFlutuante)); // false

// Formatando número de acordo com as casas decimais
console.log(numeroPontoFlutuante.toFixed(2)); // "3.46" (arredonda para 2 casas)
console.log(numeroPontoFlutuante2.toFixed(2)); // "3.45"
console.log(numeroPontoFlutuante.toFixed()); // "3" (0 casas decimais, arredondado)

// Formatando número com precisão específica
console.log(numeroPontoFlutuante.toPrecision(4)); // "3.457" (4 dígitos totais)
console.log(inteiro.toPrecision(1)); // "4e+1" (notação científica)
console.log(inteiro.toPrecision(2)); // "42" (2 dígitos totais)
console.log(inteiro.toPrecision(3)); // "42.0" (3 dígitos totais)
```

Boas Práticas ao Trabalhar com Number

- **Escolha o Método Correto:** UsetoFixed para formatação de casas decimais e toPrecision para precisão total.
- **Verifique Características:** Use Number.isInteger para garantir que o número é inteiro antes de realizar operações específicas.
- **Cuidado com Strings:** Métodos comotoFixed retornam strings; converta de volta para Number se necessário.

```
let num = parseFloat(numeroPontoFlutuante.toFixed(2));
```

Resumo

- Number.isInteger: Verifica se o número é inteiro.
- .toFixed: Controla as casas decimais exibidas.
- .toPrecision: Define a precisão total do número, ajustando dígitos ou usando notação científica.

Esses métodos são essenciais para manipulação precisa e formatação numérica em Javascript.

Nesta aula, vamos aprofundar o entendimento sobre os métodos nativos do tipo Number em JavaScript. Focaremos nas conversões entre tipos numéricos e strings, além de explorar como os números podem ser representados em diferentes bases numéricas.

Conceitos Fundamentais

- **Por que Precisamos de Conversões?**
 - Dados numéricos são frequentemente recebidos como strings, especialmente de fontes externas (formulários, APIs, etc.).
 - Métodos de conversão ajudam a transformar esses dados em números utilizáveis em cálculos e operações.
- **Bases Numéricas e Representações**
 - Números podem ser representados em diferentes bases, como decimal (base 10), binário (base 2), octal (base 8), ou hexadecimal (base 16).
 - Essa flexibilidade é útil em áreas como programação de baixo nível, criptografia e computação.



Métodos Utilizados

1. **Number.parseFloat(value)**

Converte uma string em um número de ponto flutuante.

- Ignora caracteres não numéricos após o número válido.
- Retorna NaN se não encontrar um número válido no início da string.

2. **Number.parseInt(value, radix)**

Converte uma string em um número inteiro.

- **radix** (opcional): Especifica a base numérica (padrão é 10).
- Ignora caracteres não numéricos após o número válido.
- Retorna NaN se não encontrar um número válido no início da string.

3. **.toString(radix)**

Converte um número para uma string.

- **radix** (opcional): Especifica a base numérica para conversão (padrão é 10).
- Suporta bases de 2 a 36.

Explicação do Código

- **Number.parseFloat**

- Transforma a string "32.7659" em um número decimal.
- O tipo original (string) se torna um número (number).

- **.toString com Base Numérica**

- numero.toString(2) converte o número decimal 42 para a base binária: "101010".
- numero.toString(8) converte para a base octal: "52".
- O padrão (ou 10) mantém o número em base decimal.

- **Number.parseInt**

- Converte a string para um número inteiro.
- Quando usado com um radix (base), tenta interpretar a string como um número na base fornecida.

- **Formatação de Casas Decimais**

- O método .toFixed(2) limita o número de casas decimais exibidas a 2, arredondando o valor conforme necessário.

```
let flutuanteString = "32.7659";
let inteiroString = "42";

console.log(typeof flutuanteString);

//Converter o valor (geralmente string) para um número
console.log(typeof Number.parseFloat(flutuanteString));
console.log(Number.parseFloat(inteiroString));

// Converter um inteiro para uma string - opcional - base numérica
let numero = 42; // base decimal
console.log(numero.toString()); // base decimal 42
console.log(numero.toString(10)); // base decimal 42
console.log(numero.toString(2)); //base binária 101010
console.log(numero.toString(8)); // base octal 52
// 2 - 36

//Converter uma string para um número inteiro, considerando a base numérica (opcional)
console.log(Number.parseInt(flutuanteString, 16));
console.log(Number.parseInt(inteiroString));

console.log(Number.parseFloat(flutuanteString).toFixed(2));
```

Casos de Uso Práticos

- **Entrada de Dados**
 - Um formulário envia valores como strings; use Number.parseFloat ou Number.parseInt para convertê-los.
- **Representação Binária, Octal ou Hexadecimal**
 - Útil para programadores que trabalham com redes, sistemas operacionais ou criptografia.
- **Formatação para Exibição**
 - .toFixed é ideal para exibir valores monetários ou limitar casas decimais em gráficos.

Boas Práticas ao Usar Métodos de Conversão

- **Verifique o Tipo Antes de Converter**
 - Evite erros usando typeof para checar o tipo da variável antes da conversão.
- **Entenda o radix no parseInt**
 - Sempre especifique a base para evitar interpretações incorretas.
 - Por exemplo, sem o radix, strings iniciadas com 0x podem ser interpretadas como números hexadecimais.
- **Trabalhe com Precisão**
 - Use .toFixed e .toPrecision para controlar como os números são exibidos em interfaces de usuário.

Nesta aula, vamos explorar o comportamento especial do JavaScript em situações que envolvem valores como NaN (Not a Number) e Infinity. Vamos entender como e por que esses valores aparecem, além de como usar os métodos nativos para lidar com eles de maneira eficaz.

Conceitos Fundamentais

1. O que é NaN?

- Representa um valor que **não é um número válido**, mas que tecnicamente pertence ao tipo Number.
- Surge quando realizamos operações matemáticas inválidas, como dividir 0 / 0 ou tentar converter uma string não numérica para um número.

2. O que é Infinity e -Infinity?

- Representam valores infinitamente grandes (Infinity) ou infinitamente pequenos (-Infinity).
- Podem surgir ao dividir um número por 0, exceder o maior valor representável por um número (Number.MAX_VALUE), ou em operações matemáticas que levam a valores muito grandes.

3. Métodos de Validação

- **Number.isNaN(value)**: Verifica se o valor é exatamente NaN.
- **Number.isFinite(value)**: Verifica se o valor é um número finito, ou seja, não é Infinity, -Infinity ou NaN.

Parte 1: NaN e Number.isNaN

```
let notANumber = NaN;
let notANumber2 = 0 / 0;
let string = Number("Olá");
let mensagem = "Olá Impressionador!";

// Verificar se os valores são NaN
console.log(Number.isNaN(notANumber)); // true
console.log(Number.isNaN(notANumber2)); // true
console.log(Number.isNaN(string)); // true
console.log(Number.isNaN(mensagem)); // false - "mensagem" é uma string
console.log(Number.isNaN(42)); // false - é um número válido
console.log(Number.isNaN(42.74637)); // false - também é um número válido
```

- Number.isNaN retorna true apenas quando o valor é estritamente igual a NaN.
- Ele não realiza coerção de tipo, por isso strings como "Olá" ou "Olá Impressionador!" não são NaN.

Parte 2: Infinity, -Infinity e Number.isFinite

```
let infinito = Infinity;
let infinito2 = -1 / 0;
let multiplicacao = Number.MAX_VALUE * 2;

// Verificar se os valores são finitos
console.log(Number.isFinite(infinity)); // false
console.log(Number.isFinite(infinity2)); // false
console.log(Number.isFinite(multiplicacao)); // false
console.log(Number.isFinite(42)); // true - número finito
console.log(Number.isFinite("Olá")); // false - não é número
console.log(Number.isFinite(notANumber)); // false - NaN não é finito
```

- Number.isFinite retorna true apenas para números que são finitos.
- Valores como Infinity, -Infinity ou NaN retornam false.
- Strings ou outros tipos também retornam false, pois não são números válidos.

Por que Esses Métodos São Úteis?

- **Validação de Dados**
 - Antes de realizar operações matemáticas, é importante verificar se os dados são válidos (`Number.isNaN`, `Number.isFinite`).
- **Evitar Comportamento Inesperado**
 - Valores como `NaN` ou `Infinity` podem gerar erros ou resultados inesperados em cálculos e exibições.
- **Melhor Controle**
 - Esses métodos permitem tratar casos especiais de forma clara e consistente.

Casos de Uso Práticos

- **Validação de Entrada do Usuário**

```
function validarEntrada(valor) {  
    if (Number.isNaN(Number(valor))) {  
        return "Entrada inválida! Por favor, insira um número.";  
    }  
    return "Entrada válida.";  
}  
console.log(validarEntrada("Olá")); // "Entrada inválida!"  
console.log(validarEntrada(42)); // "Entrada válida."
```

Evitar Operações com Infinity

```
function dividir(a, b) {  
    if (!Number.isFinite(a) || !Number.isFinite(b)) {  
        return "Erro: operação com infinito.";  
    }  
    return a / b;  
}  
  
console.log(dividir(10, 0)); // "Erro: operação com infinito."
```

- **Controle em Aplicações Matemáticas Complexas**

- Aplicações como gráficos, cálculos financeiros ou simulações físicas frequentemente lidam com números muito grandes ou complexos. Esses métodos ajudam a garantir que os valores sejam gerenciados corretamente.

Boas Práticas

Sempre Valide os Dados

- Antes de realizar cálculos, use `Number.isNaN` e `Number.isFinite` para garantir que os números são válidos.

Evite Comparações Diretas com NaN

- Comparar diretamente com NaN (ex: `value === NaN`) sempre retorna `false`. Use `Number.isNaN` em vez disso.

Lide com Infinity de Forma Clara

- Identifique e trate casos que podem resultar em `Infinity` ou `-Infinity`, como divisões por zero.

O Que é o Objeto Global?

No JavaScript, o **objeto global** é um "objeto de topo" que contém todas as variáveis, funções e objetos padrão que estão disponíveis no ambiente de execução. Em ambientes de navegador, o objeto global é denominado window, e em ambientes Node.js é chamado global. Esse objeto é automaticamente acessível em todo o código e serve como o contexto para variáveis e funções globais. Ao usar o JavaScript, você já está utilizando o objeto global sem nem perceber. Por exemplo, quando criamos uma variável fora de qualquer função, ela é automaticamente associada ao objeto global.

Propriedades e Métodos Comuns do Objeto Global

O objeto global fornece várias propriedades e métodos integrados que facilitam o desenvolvimento em JavaScript. Vamos dar uma olhada em algumas das mais comuns:

- **Funções de Aritmética e Cálculos: Math**

O objeto Math faz parte do objeto global e oferece métodos para executar operações matemáticas. Exemplo: Math.random() para gerar números aleatórios.

- **Trabalhando com Datas: Date**

O objeto Date, também global, é usado para manipular e formatar datas e horários. Exemplo: new Date() para obter a data e hora atual.

Benefícios do Objeto Global e Seus Métodos

- **Acesso Facilitado a Funções Comuns**

O objeto global oferece acesso direto a métodos essenciais, como funções matemáticas, manipulação de datas, e manipulação de temporizadores, sem precisar importar bibliotecas externas.

- **Padronização e Consistência**

O JavaScript proporciona um conjunto de funções padrão, como Math.random() ou Date.now(), que garantem que os resultados sejam consistentes em diferentes ambientes de execução.

- **Desempenho**

Métodos globais como Math e Date são otimizados e geralmente têm um desempenho melhor em comparação com funções personalizadas que você cria para realizar operações semelhantes.

Boas Práticas no Uso do Objeto Global

Embora o objeto global ofereça diversas ferramentas úteis, é importante tomar cuidado com o uso excessivo de variáveis e funções globais. Isso pode levar a:

- **Conflitos de Nome**

Quando você define variáveis globais, outras partes do código podem inadvertidamente sobrescrever ou entrar em conflito com elas.

- **Dificuldade de Manutenção**

O uso excessivo de variáveis globais pode tornar o código mais difícil de entender e manter, especialmente à medida que o projeto cresce.

Boas práticas:

- Evite criar variáveis globais desnecessárias.
- Use let, const, ou módulos para limitar o escopo das variáveis.
- Prefira encapsular funções e dados em objetos ou classes para melhorar a legibilidade e modularidade.

O objeto Math é um dos objetos globais nativos em JavaScript e contém métodos e propriedades que permitem realizar operações matemáticas avançadas de maneira fácil e eficiente. A principal vantagem de utilizar o Math é que ele já oferece uma série de funcionalidades matemáticas úteis sem que você precise implementar essas operações do zero.
Agora, vamos analisar os principais conceitos e métodos do objeto Math com base no código que você forneceu.

1. Propriedade Math.PI

O Math.PI é uma constante que representa o valor de Pi (π), utilizado em diversas operações matemáticas, especialmente em cálculos envolvendo círculos. O valor de Pi é aproximadamente **3.14159**.

```
const PI = Math.PI;  
console.log(PI); // Saída: 3.141592653589793
```

Explicação:

Essa propriedade retorna o valor de Pi, que pode ser usado em cálculos como o cálculo da área de um círculo ($A = \pi * r^2$). Por exemplo, se você estiver criando um programa para calcular áreas de círculos, o Math.PI é fundamental.

2. Método Math.sqrt() (Raiz Quadrada)

O método Math.sqrt() calcula a raiz quadrada de um número positivo. Se o número for negativo, o método retorna **NaN** (Not a Number), pois a raiz quadrada de um número negativo não é um número real.

```
const raizQuadrada = Math.sqrt(16); // 4 - raiz quadrada de um número  
console.log(raizQuadrada); // Saída: 4
```

Explicação:

O Math.sqrt(16) retorna **4**, pois a raiz quadrada de 16 é 4. Esse método é muito utilizado em cálculos envolvendo distâncias, áreas e outras operações matemáticas.

3. Método Math.pow() (Potência)

O método Math.pow() é utilizado para calcular a potência de um número, ou seja, elevar um número a um determinado expoente. O primeiro argumento é a base e o segundo é o expoente.

```
const potencia = Math.pow(2, 8); // 2^8 = 256  
console.log(potencia); // Saída: 256
```

Explicação:

Neste caso, estamos calculando 2 elevado à 8^a potência, que resulta em 256. O Math.pow() é muito útil quando você precisa fazer cálculos com exponenciação, como calcular áreas de superfícies, crescimento exponencial e outros.

4. Função Personalizada de Potência

No código, também vemos uma implementação personalizada de uma função de potência. Ela calcula a potência de um número de forma iterativa, multiplicando a base pelo resultado repetidamente.

```
const potenciaFuncao = function potencia(base, expoente) {  
    let resultado = 1;  
    for (let i = 0; i < expoente; i++) {  
        resultado *= base; // Multiplica o resultado pela base repetidamente  
    }  
    return resultado;  
};  
  
console.log(potenciaFuncao(2, 8)); // Saída: 256
```

Explicação:

A função potenciaFuncao simula o que o Math.pow() faz, mas de uma maneira mais manual. Ela multiplica a base por ela mesma expoente vezes. Embora essa função seja funcional, ela é menos eficiente que Math.pow() para números grandes, porque ela realiza a multiplicação de forma iterativa.

5. Função Personalizada para Raiz Quadrada

O cálculo da raiz quadrada é feito utilizando um método chamado "busca binária", que aproxima a raiz quadrada até uma precisão especificada. Vamos ver como isso funciona:

Explicação:

Esta função tenta encontrar a raiz quadrada de um número utilizando a técnica de **busca binária**. Ela continua refinando o intervalo de busca até que o resultado seja suficientemente preciso. Embora a função seja eficaz, o Math.sqrt() seria mais simples e mais eficiente para a maioria dos casos. A função personalizada pode ser útil em ambientes de aprendizado para entender como certos algoritmos funcionam.

```
const resultado = function raizQuadrada(num) {
    if (num < 0) {
        return NaN; // Raiz quadrada de números negativos não é real
    }

    let baixo = 0;
    let alto = num;
    let meio;
    const precisao = 0.000001; // Precisão desejada

    // Continua buscando até encontrar uma boa aproximação
    while (alto - baixo > precisao) {
        meio = (baixo + alto) / 2;

        if (meio * meio > num) {
            alto = meio; // Se o quadrado de 'meio' for maior que o número, diminui o intervalo
        } else {
            baixo = meio; // Se for menor, aumenta o intervalo
        }
    }

    return meio; // Aproximação final da raiz quadrada
};

console.log(resultado(16)); // Saída: 4 (aproximado)
```

Na segunda aula, vamos explorar alguns outros métodos úteis do objeto global Math. Os métodos que veremos são muito importantes para realizar cálculos precisos, como arredondamento e geração de números aleatórios.

1. Método Math.round(x) - Arredondamento

O método Math.round(x) arredonda um número para o inteiro mais próximo. Ele verifica a parte decimal e arredonda para o inteiro mais próximo com base na seguinte regra:

- Se o valor da parte decimal for **menor que 0.5**, ele arredonda para baixo.
- Se for **maior ou igual a 0.5**, ele arredonda para cima.

```
let numero = 3.45; // 3.45 arredonda para 3
let numero2 = 3.55; // 3.55 arredonda para 4

console.log(Math.round(numero)); // Saída: 3
console.log(Math.round(numero2)); // Saída: 4
```

Explicação: No primeiro exemplo, **3.45** é arredondado para **3** porque a parte decimal (0.45) é menor que 0.5. No segundo exemplo, **3.55** é arredondado para **4** porque a parte decimal (0.55) é maior ou igual a 0.5.

2. Método parseInt(x) - Converter para Inteiro

O parseInt(x) é um método que converte uma string ou número em um valor inteiro, cortando a parte decimal. Ele **não arredonda**, apenas remove a parte após o ponto decimal.

```
console.log(parseInt(numero)); // Saída: 3  
console.log(parseInt(numero2)); // Saída: 3
```

Explicação: O parseInt transforma **3.45** em **3** e **3.55** em **3**, sem considerar a parte decimal. Ou seja, ele descarta a parte decimal sem arredondar.

3. Método Math.random() - Geração de Números Aleatórios

O Math.random() retorna um número pseudo-aleatório entre **0 e 1** (não inclusivo de 1). Ou seja, o valor gerado será um número de ponto flutuante entre **0 (inclusive)** e **1 (exclusive)**.

```
const aleatorio = Math.random() * 100;  
console.log(parseFloat(aleatorio.toFixed(2))); // Arredonda para duas casas decimais
```

Explicação: Aqui, o Math.random() gera um número aleatório entre 0 e 1 e, em seguida, multiplicamos esse número por 100 para ter um número aleatório entre 0 e 100. Em seguida, usamos toFixed(2) para limitar o número a 2 casas decimais. O parseFloat é utilizado para garantir que o valor seja convertido para um número com ponto flutuante.

Exemplo com Number.parseFloat:

```
const aleatorio = Number.parseFloat((Math.random() * 100).toFixed(2));
console.log(aleatorio); // Saída: valor aleatório com 2 casas decimais
```

Explicação: Usando Number.parseFloat(), podemos garantir que o número gerado seja um número de ponto flutuante com precisão até 2 casas decimais. Isso é útil em situações como a criação de valores aleatórios em jogos, testes de software ou simulações.

Resumo e Benefícios dos Métodos

- **Math.round(x):** Útil quando você precisa arredondar um número para o inteiro mais próximo, seja para valores de cálculo, representações ou quando está lidando com dados em que a precisão de um número inteiro é necessária.
- **parseInt(x):** Ideal quando você deseja simplesmente remover a parte decimal de um número, sem se preocupar em arredondá-lo, como ao processar entradas de usuário ou ao converter valores de string.
- **Math.random():** Fundamental para gerar números aleatórios. Muito utilizado em jogos, sorteios, simulações e outras situações em que um valor aleatório é necessário.

Na aula de hoje, vamos explorar dois métodos do objeto global Math que são amplamente usados para encontrar o valor **mínimo** e **máximo** em uma lista de números: Math.min() e Math.max(). Além disso, vamos entender como o operador **spread (...)** pode ser útil quando trabalhamos com arrays.

1. Método Math.min() - Valor Mínimo

O método Math.min() é usado para encontrar o menor valor entre os números fornecidos. Ele recebe como argumento uma lista de números e retorna o menor valor.

```
console.log(Math.min(2, 45, 6, 87, 43)); // Saída: 2
```

Explicação: O Math.min() avalia todos os números passados como argumentos e retorna o menor valor entre eles. No exemplo, entre os números **2, 45, 6, 87** e **43**, o menor valor é **2**, então ele é retornado.

Erro comum:

```
console.log(Math.min([2, 45, 6, 87, 43])); // Saída: NaN
```

Explicação do erro: Se você passar um array diretamente para o Math.min(), como no exemplo acima, ele **não** funcionará corretamente e retornará NaN. Isso acontece porque o método espera os números como argumentos separados, não como um único array.

2. Método Math.max() - Valor Máximo

Assim como Math.min(), o método Math.max() é utilizado para encontrar o maior valor entre os números fornecidos.

```
console.log(Math.max(2, 45, 6, 87, 43)); // Saída: 87
```

Explicação: O Math.max() funciona da mesma maneira que o Math.min(), mas retorna o maior valor. No exemplo, entre os números **2, 45, 6, 87 e 43**, o maior valor é **87**, então ele é retornado.

3. Utilizando o Operador Spread (...) com Arrays

Agora, vamos aprender como utilizar o **operador spread (...)** para passar os elementos de um array como argumentos separados para Math.min() e Math.max(). O operador ... espalha os valores de um array, passando cada um como um argumento individual.

Exemplo com o operador spread:

```
let lista = [2, 45, 6, 87, 43, 101];
console.log(Math.min(...lista)); // Saída: 2
console.log(Math.max(...lista)); // Saída: 101
```

Explicação:

- O operador spread ... pega cada item da lista (array) e o passa como um argumento individual para o método Math.min() ou Math.max().
- No primeiro exemplo, Math.min(...lista) espalha os elementos do array [2, 45, 6, 87, 43, 101] e retorna o menor valor: **2**.
- No segundo exemplo, Math.max(...lista) espalha os elementos do array e retorna o maior valor: **101**.



Benefícios do Operador Spread

- **Simplificação de código:** Usar o operador spread permite que você passe arrays para funções como se fossem múltiplos argumentos, tornando o código mais limpo e comprehensível.
- **Flexibilidade:** Pode ser utilizado em várias situações, não apenas com métodos Math, mas também em funções e outros contextos em que você precise expandir os itens de um array.

Resumo

Hoje vimos como utilizar os métodos Math.min() e Math.max() para encontrar o valor mínimo e máximo em uma lista de números. Além disso, aprendemos como o operador spread (...) pode ser utilizado para passar os elementos de um array como argumentos separados, permitindo que trabalhemos de forma eficiente com listas de números.

Esses métodos são úteis em muitos cenários, como encontrar o menor ou maior número em uma série de dados ou realizar cálculos matemáticos que envolvem coleções de números.

Nesta aula, vamos aprender sobre o objeto global Date do JavaScript, suas funcionalidades e como ele se comporta, especialmente no contexto de **classe** e **instância**. Também vamos entender a diferença entre classes e objetos simples em JavaScript.

1. O que é o Objeto Date?

O Date é um objeto embutido do JavaScript utilizado para manipulação e formatação de **datas e horas**. Ele faz parte da **API de data e hora** do JavaScript e permite que você crie instâncias que representam pontos específicos no tempo.

Quando você utiliza o `new Date()`, você está criando uma **instância** da classe Date, que contém informações como o ano, mês, dia, hora, minuto e segundo do momento em que a instância foi criada.

2. Diferença entre Classe e Instância

Classe: Uma classe é uma estrutura que define as propriedades e comportamentos comuns para os objetos de uma determinada categoria. Ela pode ser considerada como uma planta, uma definição de como um objeto será.

Instância: Uma instância é a criação de um objeto real a partir de uma classe. Quando você cria um objeto de uma classe, você está criando uma instância dessa classe. Cada instância pode ter seus próprios valores para as propriedades definidas pela classe.

3. Exemplo Prático: Classe e Instância

Primeiro, vamos entender o que são **classe** e **instância** com um exemplo utilizando a classe Carro.

```
class Carro {
  marca = "Toyota";
  modelo = "Corolla";
  ano = 2024;

  ligar() {
    console.log("Carro ligado");
  }

  desligar() {
    console.log("Carro desligado!");
  }

  exibirInformacoes() {
    console.log(
      `O carro é de modelo: ${Carro.modelo} da marca ${Carro.marca}, do ano de ${Carro.ano}`
    );
  }
}
```

No código anterior:

- **Classe Carro:** Define a estrutura de um carro com propriedades como marca, modelo e ano, além de métodos para **ligar**, **desligar** e **exibir informações** sobre o carro.
- **Instância de Carro:** Para criar um carro real a partir dessa classe, você utiliza new Carro(), criando uma instância.

```
const classeCarro = new Carro();
classeCarro.exibirInformacoes(); // Exibe as informações do carro
```

Aqui, classeCarro é uma instância da classe Carro e pode acessar os métodos e propriedades definidas nela.

4. Objeto Global Date

Agora, vamos falar sobre o objeto global Date. O Date também é uma **classe** em JavaScript. Porém, ao contrário da classe Carro, a classe Date já está integrada no JavaScript, e não precisamos criá-la manualmente. Podemos instanciá-la diretamente.

Exemplo com Date:

```
const dataAtual = new Date(); // criando uma instância do objeto/classe Date
console.log(dataAtual); // Exibe a data e hora atuais
```

Aqui, dataAtual é uma **instância** da classe Date, que, ao ser criada, armazena a data e hora no momento exato de sua criação.

Por que a classe Date é útil?

A classe Date oferece uma maneira fácil de:

- Obter a data e hora atuais.
- Manipular datas, como adicionar ou subtrair dias.
- Comparar datas.
- Exibir datas em diferentes formatos.

5. Diferença entre Objeto Simples e Classe Date

- **Objeto Simples:** Um objeto simples é uma coleção de pares chave-valor. Por exemplo:

```
const carro = {  
    modelo: "Corolla",  
    marca: "Toyota",  
    ano: 2024,  
};
```

- O objeto carro armazena informações sobre um carro, mas não possui métodos internos para manipular essas informações de forma dinâmica, como a classe Date faz com datas.
- **Classe Date:** A classe Date é um tipo de objeto especial em JavaScript, que não é apenas uma coleção de dados, mas também fornece métodos para trabalhar com esses dados (como mostrar a data atual ou modificar a data).

Na aula de hoje, vamos explorar em detalhes o objeto Date do JavaScript. O Date é uma classe que lida com datas e horas. Através dessa classe, podemos criar instâncias que representam pontos específicos no tempo, e a partir disso, utilizar métodos para extrair informações sobre a data, hora, e até mesmo fazer comparações ou cálculos com datas.

1. Criando uma Instância de Date

A primeira coisa que fazemos ao trabalhar com o objeto Date é criar uma instância. Para isso, usamos o construtor `new Date()`:

```
const dataAtual = new Date(); // Criamos uma instância do objeto Date  
console.log(dataAtual); // Exibe a data e hora atuais
```

Quando chamamos `new Date()`, a instância `dataAtual` armazenará a data e hora exatas no momento em que o código for executado.

2. Métodos para Obter Partes Específicas da Data

Uma das utilidades do objeto Date é a capacidade de obter partes específicas de uma data, como o ano, mês, dia, hora, minutos e segundos. Vamos ver os principais métodos:

Método `getFullYear()`

Esse método retorna o **ano atual**:

```
console.log(dataAtual.getFullYear()); // Exibe o ano atual (exemplo: 2024)
```

Método getMonth()

Esse método retorna o **mês atual**. **Importante:** O valor retornado é entre 0 e 11, onde 0 corresponde a Janeiro e 11 a Dezembro.

```
console.log(dataAtual.getMonth()); // Exibe o mês atual (exemplo: 11 para Dezembro)
```

Método getDate()

Esse método retorna o **dia do mês** (de 1 a 31):

```
console.log(dataAtual.getDate()); // Exibe o dia atual (exemplo: 30)
```

Método getHours()

Esse método retorna a **hora atual** (de 0 a 23):

```
console.log(dataAtual.getHours()); // Exibe a hora atual (exemplo: 14 para 14:00)
```

Método getMinutes()

Esse método retorna os **minutos atuais** (de 0 a 59):

```
console.log(dataAtual.getMinutes()); // Exibe os minutos atuais (exemplo: 30)
```

Método getSeconds()

Esse método retorna os **segundos atuais** (de 0 a 59):

```
console.log(dataAtual.getSeconds()); // Exibe os segundos atuais (exemplo: 45)
```



3. O Valor do Timestamp

O **timestamp** é uma representação numérica do tempo, que expressa a quantidade de milissegundos desde a **data de referência**, que é **1 de janeiro de 1970 (1970-01-01)**, às 00:00:00 UTC.

Método `getTime()`

O método `getTime()` retorna o timestamp da instância Date:

```
let timestamp = dataAtual.getTime(); // Retorna o timestamp em milissegundos  
console.log(timestamp); // Exibe o timestamp da data atual
```

A principal vantagem de usar o timestamp é que ele nos dá um número com o qual podemos realizar cálculos, como a diferença entre duas datas.

4. Criando uma Data a Partir do Timestamp

Você também pode criar um objeto Date utilizando o **timestamp**. Por exemplo:

```
console.log(new Date(1729883324187)); // Cria uma nova data a partir do timestamp  
console.log(new Date(1729883373425)); // Cria uma nova data a partir de outro timestamp
```

Cada timestamp corresponde a uma data específica. Nesse exemplo, o número 1729883324187 representa uma data e hora em milissegundos desde 1 de janeiro de 1970.

5. Criando uma Data a Partir de uma String

Você pode criar uma data a partir de uma **string de data**, usando um formato que o JavaScript reconheça:

```
let agora = new Date("2024-10-07"); // Cria uma data a partir de uma string
console.log(agora.getMonth() + 1); // Exibe o mês (Lembrando que o mês começa do 0,
```

Aqui, a string "2024-10-07" representa uma data no formato "ano-mês-dia". O método `getMonth()` retornará o valor 9, mas para exibir o mês corretamente, somamos 1, pois os meses começam de 0.

6. Resumo dos Métodos Importantes

- **`getFullYear()`**: Retorna o ano da instância.
- **`getMonth()`**: Retorna o mês da instância (0 a 11).
- **`getDate()`**: Retorna o dia do mês.
- **`getHours()`**: Retorna a hora.
- **`getMinutes()`**: Retorna os minutos.
- **`getSeconds()`**: Retorna os segundos.
- **`getTime()`**: Retorna o timestamp (milissegundos desde 1 de janeiro de 1970).
- **Criação de data via `new Date(timestamp)`**: Permite criar uma data a partir de um timestamp.
- **Criação de data via string**: Permite criar uma data a partir de uma string de formato reconhecido pelo JavaScript.

Conclusão

Hoje aprendemos sobre o objeto global Date e seus métodos para manipular datas e horas em JavaScript. Vimos como obter diferentes partes de uma data, como ano, mês, dia, hora, minuto e segundo. Além disso, exploramos o conceito de **timestamp** e como podemos usar o número de milissegundos desde a data de referência para representar datas de forma numérica.